设计模式包教不包会

wizardforcel

Published with GitBook



目錄

design pattern 包教不包会

> 作者：AlfredTheBest
>
> 来源：Design-Pattern

课程列表

设计模式

在软件工程中，设计模式（design pattern）是对软件设计中普遍存在（反复出现）的各种问题，所提出的解决方案。这个术语是由埃里希·伽玛（Erich Gamma）等人在1990年代从建筑设计领域引入到计算机科学的。

设计模式并不直接用来完成代码的编写，而是描述在各种不同情况下，要怎么解决问题的一种方案。

面向对象设计模式通常以类或对象来描述其中的关系和相互作用，但不涉及用来完成应用程序的特定类或对象。设计模式能使不稳定依赖于相对稳定、具体依赖于相对抽象，避免会引起麻烦的紧耦合，以增强软件设计面对并适应变化的能力。

并非所有的软件模式都是设计模式，设计模式特指软件“设计”层次上的问题。还有其它非设计模式的模式，如架构模式。同时，算法不能算是一种设计模式，因为算法主要是用来解决计算上的问题，而非设计上的问题。

肯特·贝克和沃德·坎宁安在1987年，利用克里斯托佛·亚历山大在建筑设计领域里的思想开发了设计模式并把此思想应用在Smalltalk中的图形用户接口（GUI）的生成中。一年后埃里希·伽玛在他的苏黎世大学博士毕业论文中开始尝试把这种思想改写为适用于软件开发。与此同时James Coplien 在1989年至1991年也在利用相同的思想致力于C++的开发，而后于1991年发表了他的著作Advanced C++ Programming Styles and Idioms。同年Erich Gamma 得到了博士学位，然后去了美国，在那与Richard Helm, Ralph Johnson ,John Vlissides 合作出版了《设计模式：可复用面向对象软件的基础》（Design Patterns - Elements of Reusable Object-Oriented Software） 一书，在此书中共收录了23个设计模式。

这四位作者在软件开发领域里以“四人帮”（英语，Gang of Four，简称GoF）而闻名，并且他们在此书中的协作导致了软件设计模式的突破。有时，GoF也会用于代指《设计模式》这本书。

《设计模式》一书原先把设计模式分为创建型模式、结构型模式、行为型模式，把它们通过授权、聚合、诊断的概念来描述。若想更进一步了解关于面向对象设计的背景，参考接口模式、内聚。若想更进一步了解关于面向对象编程的背景，参考继承，接口，多态。

简单工厂模式

活字印刷 面向对象

话说三国时期，曹操带领百万大军攻打东吴，大军在长江赤壁驻扎，军船连成一片，眼看就要灭掉东吴，统一天下，曹操大悦，于是大宴众文武，在酒席间，曹操诗兴大发，不觉吟道：喝酒唱歌，人生真爽。众文武齐呼："丞相好诗！于是一臣子速命印刷工匠刻板印刷，以便流传天下。"

样张出来给曹操一看，曹操感觉不妥，说到："喝与唱，此话过俗，应该为'对酒当歌'较好！"，于是此臣就命工匠重新来过。工匠眼看连夜刻板之工，彻底白费，心中叫苦不迭。只得照办。"

样张再次出来请曹操过目，曹操细细一品，感觉还是不好，说："人生真爽太过直接，应改问语才够意境，因此应改为'对酒当歌，人生几何？'当臣子转告工匠之时，工匠晕倒！"

为何三国时期的工匠有如此的问题？

当时活字印刷还未发明，所以要改字的时候必须要整个刻板重刻。如果有活字印刷，则只需更改四个字就可，其余工作都未白做，岂不妙哉。

- 要改，只需要更改之字，此为可维护。
- 这写字并非用完这次就无用，完全可以在后来的印刷中重复使用，此乃可复用。
- 此诗若要加字，只需另刻字加入即可，这是可扩展。
- 字的排列其实可能是竖排也可能是横排，此时只需要将活字移动就可以做到满足排列需求，此是灵活性好

面对对象的分析设计编程思想，通过封装，继承多态把程序的耦合度降低，传统印刷术的问题就在于所有的字都刻在同一版面上造成耦合度太高所致，开始用设计模式使得程序更加灵活，易于修改，易于复用。

简单工厂模式

例：计算器，到底要实例化谁，将来会不会增加实例化的对象，把很容易变化的地方用一个单独的类来做这个创造实例的过程，这个就是工厂。 简单的运算工厂类

```
public class OperationFactory
{
    public static operation createOperate(string operate)
    {
    Operation oper = null;
    switch (operate)
    {
        case "+":
            oper = new OperationAdd();
            break;
        case "-":
            oper = new OperationSub();
            break;
        case "*":
            oper = new OperationMul();
            break;
        case "/":
            oper = new OperationDiv();
            break;
    }
    return oper;
    }
}
```

客户端代码

```
Operation oper;
oper = OperationFactory.createOperate("+");
oper.NumberA = 1;
oper.NumberB = 2;
double result = oper.GetResult();
```

这样，以后需要增加各种复杂运算，比如平方根，立方根，自然对数等等，只要增加相对应的运算子类就可以了。

策略模式

应用场景

一个商场收银软件，营业员根据客户所购买的商品的单价和数量，向客户收费

用两个文本框来输入单价和数量，一个确定按钮来算出每种商品的费用，用个列表框来记录商品的清单，一个标签来记录总计，一个重置按钮来重新开始。

```
double total = 0.0d;
private void btn0k_Click(object sender, EventArgs e)
{
    double totalPrices = Convert.ToDouble(txtPrice.Text) * Convert
    total = total + totalPrices;
    IbxList.Items.Add("单价：" +txtPrice.Text + "数量：" ＋ txtNum.T
    IblResult.Text = total.ToString();
}
```

比如遇到节假日 增加打折

```
double total = 0.0d;

private void Form_Load(object sender, EventArgs e)
{
    cbxType.Items.AddRange(new object[] {"正常收费","打八折","打五折"
    cbxType.SlectedIndex = 0;
}

private void btn0k_Click(object sender, EventArgs e)
{
double totalPrices = 0.0d;
switch(cbxType.Selectedindex)
{
    case 0:
             totalPrices = Convert.ToDouble(txtPrice.Text) * Conve
    case 1:
             totalPrices = Convert.ToDouble(txtPrice.Text) * Conve
    case 2:
             totalPrices = Convert.ToDouble(txtPrice.Text) * Conve
}
   total = total + totalPrices;
    IbxList.Items.Add("单价：" +txtPrice.Text + "数量：" ＋ txtNum.T
    IblResult.Text = total.ToString();
}
```

简单工厂实现

面对对象的编程，并不是类越多越好，类的划分是为了封装，但是分装的基础是抽象，具有相同属性和功能的对象的抽象集合才是类。打一折和打九折只是形式的不同，抽象分析出来，所有的打折算法都是一样的，所以打折算法应该是一个类。

```
//现金收费抽象类
abstract class CashSuper
{
    public abstract double acceptCash(double money);
}

//正常收费子类
class CashNormal: CashSuper
{
    public override double acceptCash(double money)
    {
        return money;
    }
}

//打折收费子类
class CashRebate: CashSuper
```

```
{
    private double moneyRebate = 1d;
    public Cash Rebebate(string moneyRebate)
    {
        this.moneyRebate = double.Parse(moneyRebate);
    }
    publci override double acceptCash(double money)
    {
        return money * moneyRebate;
    }
}

//返利收费子类
class cashReturn: CashSuper
{
    private double moneyCondition = 0.0d;
    private double moneyreturn = 0.0d;
    public CashReturn(string moneyCondition, string moneyReturn)
    {
        this.moneyCondition = double.Parse(moneyCondtion);
        this.moneyReturn = double.Parse(moneyReturn);
    }

    public override double acceptCash(double money)
    {
        double result = money;
        if(money >= moneyCondition)
        {
            result = money - Math.Floor(money / moneyCondition) * r
        }
            return result;
    }

}

//现金收费工厂类
class CashFactory
{
    public static CashSuper createCashAccept(string type)
    {
        CashSuper cs = null;
        switch (type)
        {
            case:"正常收费":
                cs = new CashNormal();
                break;
            case:"满300返100":
                CashReturn cr1 = new CashReturn("300","100");
                cs = cr1;
                break;
            case:"打8折":
                CashRebate cr2 = new CashRebate("0,8");
                cs = cr2;
```

```
                    break;
            }
            return cs;
        }
    }

    //客户端程序主要部分
    double total = 0.0d;
    private void btn0k_clik(object sneder, EventArgs e)
    {
        cashSuper csuper = CashFactory.createCashAccept(cbxType.Selecte
        double totalPrices = 0d;
        totalPrices = csuper.acceptCash(Convert.ToDouble(txtPrice.txt)
        total = total + totalPrices;
        IbxList.Items.Add("单价：" +txtPrice.Text + "数量：" ＋ txtNum.Te
         IblResult.Text = total.ToString();

    }
```

简单工厂模式虽然能解决这个问题，但是这个模式知识解决对象的创建问题，而且由于工厂本身包括了所有的收费方式，商场是可能经常性地更改打折额度和返利额度，每次维护或扩展收费方式都要改动这个工厂，以致代码需重新编译部署，这真的是很糟糕的处理方式，所以用它不是做好的办法，。面对算法的时常变动应该有更好的办法。

策略模式

策略模式定义了算法家族，分别封装起来，让它们之间可以互相替换，此模式让算法的变化，不会影响到使用算法的客户。

```
//CashContext类
class CashContext
{
    private CashSuper cs;
    public CashContext(CashSuper csuper)
    {
        this.cs = csuper;
    }

    public double GetResult(double money)
    {
        return cs.accptCash(money);
    }

}

//客户端主要代码
double total = 0.0d;
private void btn0k_Click(object sender, EventArgs e)
{
    CashContext cc = null;
    switch (cbxType.selectedItem.ToString())
    {
        case: "正常收费":
            cc = new CashContext(new CashNormal());
            break;
        case: "满300返100":
            cc = new CashContext(new CashReturn ("300","100"));
            break;
        case: "打8折":
            cc = new CashContext(new CashRebate("0.8"));
            break;
    }
    double totalPrices = 0d;
    totalPrices = cc.GetResult(convert.ToDouble(txtPrice.Text)*conv
    total = total + totalPrices;
    IbxList.Items.Add("单价：" +txtPrice.Text + "数量：" ＋ txtNum.Te
     IblResult.Text = total.ToString();

}
```

虽然策略模式写出来了，但是不应该让客户端去判断用哪一个算法。

策略与简单工厂结合

```
class CashContext
{
    CashSuper cs  = null;

    public CashContext(String type)
    {
        switch(type)
        {
            case "正常收费":
                CashNormal cs0 = new CashNormal();
                cs = cs0;
                break;
            case "满300返100":
                CashReturn cr1 = new CashReturn("300","100");
                cs = cr1;
                break;
            case "打8折"：
                CashReturn cr2 = new CashRebate("0.8");
                cs = cr2;
                break;
        }
        public double GetResult(double money)
        {
            return cs.acceptCash(money);
        }
    }
}

//客户端代码
double total = 0.0d;
private void btn0k_Click(object sender, EvnetArgs e)
{
    CashContext csuper = new CashContext(cbxType.slectedItem.ToStri
    double totalPrices = 0d;
    totalPrices = csuper.GetResult(Convert.ToDouble(txtPrice.Text)
    total = total + totalPrices;
    IbxList.Items.Add("单价：" +txtPrice.Text + "数量：" ＋ txtNum.Te
     IblResult.Text = total.ToString();
}

//简单工厂模式的用法
CashSuper csuper = CashFactory.CreateCashAccept(cbxType.SelectedIte
=csuper.GetResult;

//策略模式与简单工厂结合的用法
CashContext csuper = new (CashContext(cbxType.SelectedItem.ToStrin
=csuper.GetResult;
```

简单工厂模式让客户端认识两个类， CashSuper和CashFactory,而策略模式与简单工厂结合的用法，客户端只需要认识一个类CashContext就可以了。耦合更加降低。

策略模式解析

策略模式是一种定义一系列算法的方法，从概念上来看，所有这些算法完成的都是相通的工作，只是实现不同，它可以以相同的方式调用所有的算法，减少各种算法类和使用算法类之间的耦合。 策略模式就是用来封装算法的，但是实践中，我们发现可以用它来分装几乎任何类型的规则，只要在分析过程中听到需要在不同的时间应用不同的业务规则，就可以考虑使用策略模式处理这种变化的可能性。

面向对象六大原则

概述

在工作初期，我们可能会经常会有这样的感觉，自己的代码接口设计混乱、代码耦合较为严重、一个类的代码过多等等，自己回头看的时候都觉得汗颜。再看那些知名的开源库，它们大多有着整洁的代码、清晰简单的接口、职责单一的类，这个时候我们通常会捶胸顿足而感叹：什么时候老夫才能写出这样的代码！

相关资料

iOS

AFNetworking2.0源码解析

AFNetworking源码

单一原则(Single Responsibility Principle)

简述

SRP:就一个类而言，应该仅有一个引起它变化的原因。 单一职责的划分界限并不是如马路上的行车道那么清晰，很多时候都是需要靠个人经验来界定。当然最大的问题就是对职责的定义，什么是类的职责，以及怎么划分类的职责。如果一个类承担的职责过多，就等于把这些职责耦合在一起，一个职责的变化可能会削弱或者抑制这个类完成其他职责的能力。这种耦合会导致脆弱的设计，当变化发生时，设计会遭受到意想不到的破坏。

当然，软件设计真正要做的许多内容，就是发现职责并把那些职责相互分离，就是抽象的能力。其实要去判断是否应该分离出类来，也不难，那就是如果你能想到多于一个的动机去改变一个类，那么这个类就具有多于一个的职责。

示例

```
/**
* An HTTP stack abstraction.
*/
public interface HttpStack {
 /**
 * 执行Http请求,并且返回一个HttpResponse
 */
public HttpResponse performRequest(Request<?> request,         Map<
    throws IOException, AuthFailureError;
}
```

HttpStack中这个函数的职责就是执行网络请求并且返回一个Response。它的职责很单一，这样在需要修改执行网络请求的相关代码时我们只需要修改实现HttpStack接口的类，而不会影响其它的类的代码。如果某个类的职责包含有执行网络请求、解析网络请求、进行gzip压缩、封装请求参数等等，那么在你修改某处代码时你就必须谨慎，以免修改的代码影响了其它的功能。但是当职责单一的时候，你修改的代码能够基本上不影响其它的功能。这就在一定程度上保证了代码的可维护性。注意，单一职责原则并不是说一个类只有一个函数，而是说这个类中的函数所做的工作必须要是高度相关的，也就是高内聚。

里氏替换原则(Liskov Substitution Principle)

简述

肯定有不少人跟我刚看到这项原则的时候一样，对这个原则的名字充满疑惑。其实原因就是这项原则最早是在1988年，由麻省理工学院的一位姓里的女士（Barbara Liskov）提出来的。

定义1：如果对每一个类型为 T1的对象 o1，都有类型为 T2 的对象o2，使得以 T1定义的所有程序 P 在所有的对象 o1 都代换成 o2 时，程序 P 的行为没有发生变化，那么类型 T2 是类型 T1 的子类型。

定义2：所有引用基类的地方必须能透明地使用其子类的对象。

问题由来：有一功能P1，由类A完成。现需要将功能P1进行扩展，扩展后的功能为P，其中P由原有功能P1与新功能P2组成。新功能P由类A的子类B来完成，则子类B在完成新功能P2的同时，有可能会导致原有功能P1发生故障。

解决方案：当使用继承时，遵循里氏替换原则。类B继承类A时，除添加新的方法完成新增功能P2外，尽量不要重写父类A的方法，也尽量不要重载父类A的方法。

继承包含这样一层含义：父类中凡是已经实现好的方法（相对于抽象方法而言），实际上是在设定一系列的规范和契约，虽然它不强制要求所有的子类必须遵从这些契约，但是如果子类对这些非抽象方法任意修改，就会对整个继承体系造成破坏。而里氏替换原则就是表达了这一层含义。

继承作为面向对象三大特性之一，在给程序设计带来巨大便利的同时，也带来了弊端。比如使用继承会给程序带来侵入性，程序的可移植性降低，增加了对象间的耦合性，如果一个类被其他的类所继承，则当这个类需要修改时，必须考虑到所有的子类，并且父类修改后，所有涉及到子类的功能都有可能会产生故障。

里氏替换原则通俗的来讲就是：子类可以扩展父类的功能，但不能改变父类原有的功能。它包含以下4层含义：

- 子类可以实现父类的抽象方法，但不能覆盖父类的非抽象方法。
- 子类中可以增加自己特有的方法。
- 当子类的方法重载父类的方法时，方法的前置条件（即方法的形参）要比父类方法的输入参数更宽松。
- 当子类的方法实现父类的抽象方法时，方法的后置条件（即方法的返回值）要比父类更严格。

示例

还是以HttpStack为例，Volley定义了HttpStack来表示执行网络请求这个抽象概念。在执行网络请求时，我们只需要定义一个HttpStack对象，然后调用performRequest即可。至于HttpStack的具体实现由更高层的调用者给出。示例如下：

```
  public static RequestQueue newRequestQueue(Context context, HttpS
      File cacheDir = new File(context.getCacheDir(), DEFAULT_CACHE
      String userAgent = "volley/0";
        // 代码省略
// 1、构造HttpStack对象
      if (stack == null) {
          if (Build.VERSION.SDK_INT >= 9) {
              stack = new HurlStack();
          } else {
              // Prior to Gingerbread, HttpUrlConnection was unreli
              // See: http://android-developers.blogspot.com/2011/0
              stack = new HttpClientStack(AndroidHttpClient.newInst
          }
      }
// 2、将HttpStack对象传递给Network对象
      Network network = new BasicNetwork(stack);
// 3、将network对象传递给网络请求队列
      RequestQueue queue = new RequestQueue(new DiskBasedCache(cach
      queue.start();
      return queue;
  }
```

BasicNetwork的代码如下:

```
/**
 * A network performing Volley requests over an {@link HttpStack}.
 */
public class BasicNetwork implements Network {
    // HttpStack抽象对象
    protected final HttpStack mHttpStack;
    protected final ByteArrayPool mPool;
    public BasicNetwork(HttpStack httpStack) {
        this(httpStack, new ByteArrayPool(DEFAULT_POOL_SIZE));
    }


    public BasicNetwork(HttpStack httpStack, ByteArrayPool pool) {
        mHttpStack = httpStack;
        mPool = pool;
    }
}
```

上述代码中，BasicNetwork构造函数依赖的是HttpStack抽象接口，任何实现了HttpStack接口的类型都可以作为参数传递给BasicNetwork用以执行网络请求。这就是所谓的里氏替换原则，任何父类出现的地方子类都可以出现，这不就保证了可扩展性吗？

减少LSP妨碍

契约（Contracts）

处理 LSP 过分妨碍的一个策略是使用契约，契约清单有 2 种形式：执行说明书（executable specifications）和错误处理，在执行说明书里，一个详细类库的契约也包括一组自动化测试，而错误处理是在代码里直接处理的，例如在前置条件，后置条件，常量检查等，可以从 Bertrand Miller 的大作《契约设计》中查看这个技术。虽然自动化测试和契约设计不在本篇文字的范围内，但当我们用的时候我还是推荐如下内容：

检查使用测试驱动开发（Test-Driven Development）来指导你代码的设计 设计可重用类库的时候可随意使用契约设计技术 对于你自己要维护和实现的代码，使用契约设计趋向于添加很多不必要的代码，如果你要控制输入，添加测试是非常有必要的，如果你是类库作者，使用契约设计，你要注意不正确的使用方法以及让你的用户使之作为一个测试工具。 契约（Contracts）

处理 LSP 过分妨碍的一个策略是使用契约，契约清单有 2 种形式：执行说明书（executable specifications）和错误处理，在执行说明书里，一个详细类库的契约也包括一组自动化测试，而错误处理是在代码里直接处理的，例如在前置条件，后置条件，常量检查等，可以从 Bertrand Miller 的大作《契约设计》中查看这个技术。虽然自动化测试和契约设计不在本篇文字的范围内，但当我们用的时候我还是推荐如下内容：

检查使用测试驱动开发（Test-Driven Development）来指导你代码的设计 设计可重用类库的时候可随意使用契约设计技术 对于你自己要维护和实现的代码，使用契约设计趋向于添加很多不必要的代码，如果你要控制输入，添加测试是非常有必要的，如果你是类库作者，使用契约设计，你要注意不正确的使用方法以及让你的用户使之作为一个测试工具。

避免继承

避免 LSP 妨碍的另外一个测试是：如果可能的话，尽量不用继承，在Gamma的大作《Design Patterns – Elements of Reusable Object-Orineted Software》中，我们可以看到如下建议：

Favor object composition over class inheritance

尽量使用对象组合而不是类继承 有些书里讨论了组合比继承好的唯一作用是静态类型，基于类的语言（例如，在运行时可以改变行为），与 JavaScript 相关的一个问题是耦合，当使用继承的时候，继承子类型和他们的基类型耦合在一起了，就是说基类型的改变会影响到继承子类型。组合倾向于对象更小化，更容易向静态和动态语言语言维护。

与行为有关，而不是继承 到现在，我们讨论了和继承上下文在内的里氏替换原则，指示出 JavaScript 的面向对象实。不过，里氏替换原则（LSP）的本质不是眞的和继承有关，而是行为兼容性。JavaScript 是一个动态语言，一个对象的契约行为不是对象的类型决定的，而是对象期望的功能决定的。里氏替换原则的初始构想是作为继承的一个原则指南，等价于对象设计中的隐式接口。

依赖倒置原则(Dependence Inversion Principle)

应用场景

电脑在以前维修的话是根本不可能的事，可是现在却特别容易，比如说内存坏了，买个内存条，硬盘坏了，买个硬盘换上。为啥这么方便？从修电脑里面就有面相对象的几大设计原则，比如单一职责原则，内存坏了，不应该成为更换CPU的理由，它们各自的职责是明确的。再比如开放—封闭原则，内存不够只要插槽足够就可以添加。还有依赖倒转原则，原话解释是抽象不应该依赖细节，细节应该依赖于抽象，说白了，就是要针对接口编程，不要对实现编程，无论主板，CPU，内存，硬盘都是针对接口设计的，如果是针对实现来设计，内存就要对应的某个品牌的主板，那就会出现换内存需要把主板也换了的尴尬。

为什么叫反转呢？

面对过程开发时，为了使得常用代码可以复用，一般都会把这些常用代码写成许许多多函数的程序库，这样我们做新项目时，去调用这些底层的函数就可以了。比如我们做的项目大多要访问数据库，所以我们就把访问数据库的代码写成了函数，每次做新项目时就去调用，这就叫做高层模块依赖底层模块。

但是要做新项目是 业务逻辑的高层模块都是一样的，客户却希望使用不同的数据库或存储信息方式，这时出现麻烦了。我们希望能再次利用这些高层模块，但高层模块都是与底层的访问数据库绑定在一起的，没办法复用这些高层模块，这就非常糟糕了。就像刚才说的，PC里如果CPU，内存，硬盘都是需要依赖具体的主板，主板一坏，所有的部件都没法用了，显然不合理，而如果不管高层模块还是底层模块，它们都依赖于抽象，具体一点就是接口或者抽象类，只要接口是稳定的，那么任何一个的更改都不用担心其它受影响，这就使得无论高层模块还是底层模块都可以很容易被复用，这才是最好的办法。

实例

依赖倒置原则基于这样一个事实：相对于细节的多变性，抽象的东西要稳定的多。以抽象为基础搭建起来的架构比以细节为基础搭建起来的架构要稳定的多。在java中，抽象指的是接口或者抽象类，细节就是具体的实现类，使用接口或者抽象类的目的是制定好规范和契约，而不去涉及任何具体的操作，把展现细节的任务交给他们的实现类去完成。

依赖倒置原则的核心思想是面向接口编程，我们依旧用一个例子来说明面向接口编程比相对于面向实现编程好在什么地方。场景是这样的，母亲给孩子讲故事，只要给她一本书，她就可以照着书给孩子讲故事了。代码如下：

```
class Book{
    public String getContent(){
        return "很久很久以前有一个阿拉伯的故事……";
    }
}

class Mother{
    public void narrate(Book book){
        System.out.println("妈妈开始讲故事");
        System.out.println(book.getContent());
    }
}

public class Client{
    public static void main(String[] args){
        Mother mother = new Mother();
        mother.narrate(new Book());
    }
}
```

运行良好，假如有一天，需求变成这样：不是给书而是给一份报纸，让这位母亲讲一下报纸上的故事，报纸的代码如下：

```
class Newspaper{
    public String getContent(){
        return "林书豪38+7领导尼克斯击败湖人……";
    }
}
```

这位母亲却办不到，因为她居然不会读报纸上的故事，这太荒唐了，只是将书换成报纸，居然必须要修改Mother才能读。假如以后需求换成杂志呢？换成网页呢？还要不断地修改Mother，这显然不是好的设计。原因就是Mother与Book之间的耦合性太高了，必须降低他们之间的耦合度才行。 我们引入一个抽象的接口IReader。读物，只要是带字的都属于读物：

```
interface IReader{
    public String getContent();
}
```

Mother类与接口IReader发生依赖关系，而Book和Newspaper都属于读物的范畴，他们各自都去实现IReader接口，这样就符合依赖倒置原则了，代码修改为：

```
class Newspaper implements IReader {
    public String getContent(){
        return "林书豪17+9助尼克斯击败老鹰……";
    }
}
class Book implements IReader{
    public String getContent(){
        return "很久很久以前有一个阿拉伯的故事……";
    }
}

class Mother{
    public void narrate(IReader reader){
        System.out.println("妈妈开始讲故事");
        System.out.println(reader.getContent());
    }
}

public class Client{
    public static void main(String[] args){
        Mother mother = new Mother();
        mother.narrate(new Book());
        mother.narrate(new Newspaper());
    }
}
```

这样修改后，无论以后怎样扩展Client类，都不需要再修改Mother类了。这只是一个简单的例子，实际情况中，代表高层模块的Mother类将负责完成主要的业务逻辑，一旦需要对它进行修改，引入错误的风险极大。所以遵循依赖倒置原则可以降低类之间的耦合性，提高系统的稳定性，降低修改程序造成的风险。

采用依赖倒置原则给多人并行开发带来了极大的便利，比如上例中，原本Mother类与Book类直接耦合时，Mother类必须等Book类编码完成后才可以进行编码，因为Mother类依赖于Book类。修改后的程序则可以同时开工，互不影响，因为Mother与Book类一点关系也没有。参与协作开发的人越多、项目越庞大，采用依赖导致原则的意义就越重大。现在很流行的TDD开发模式就是依赖倒置原则最成功的应用。

传递依赖关系有三种方式，以上的例子中使用的方法是接口传递，另外还有两种传递方式：构造方法传递和setter方法传递，相信用过Spring框架的，对依赖的传递方式一定不会陌生。 在实际编程中，我们一般需要做到如下3点：

- 低层模块尽量都要有抽象类或接口，或者两者都有。
- 变量的声明类型尽量是抽象类或接口。
- 使用继承时遵循里氏替换原则。 依赖倒置原则的核心就是要我们面向接口编程，理解了面向接口编程，也就理解了依赖倒置。

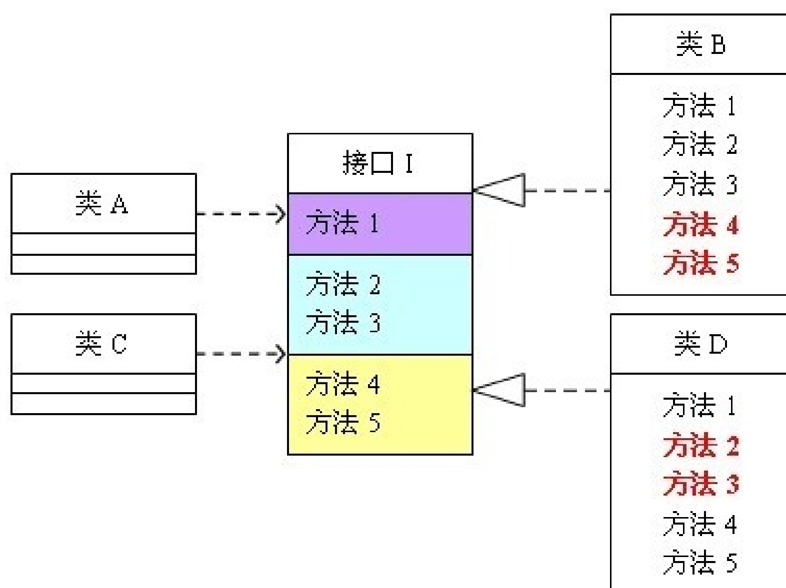
接口隔离原则(Interface Segregation Principle)

简介

接口隔离原则（英语：interface-segregation principles， 缩写：ISP）指明没有客户(client)应该被迫依赖于它不使用方法。接口隔离原则(ISP)拆分非常庞大臃肿的接口成为更小的和更具体的接口，这样客户将会只需要知道他们感兴趣的方法。这种缩小的接口也被称为角色接口（role interfaces）。接口隔离原则(ISP)的目的是系统解开耦合，从而容易重构，更改和重新部署。接口隔离原则是在SOLID (面向对象设计)中五个面向对象设计(OOD)的原则之一，类似于在GRASP (面向对象设计)中的高内聚性。

实例

定义：客户端不应该依赖它不需要的接口；一个类对另一个类的依赖应该建立在最小的接口上。 问题由来：类A通过接口I依赖类B，类C通过接口I依赖类D，如果接口I对于类A和类B来说不是最小接口，则类B和类D必须去实现他们不需要的方法。 解决方案：将臃肿的接口I拆分为独立的几个接口，类A和类C分别与他们需要的接口建立依赖关系。也就是采用接口隔离原则。 举例来说明接口隔离原则：



这个图的意思是：类A依赖接口I中的方法1、方法2、方法3，类B是对类A依赖的实现。类C依赖接口I中的方法1、方法4、方法5，类D是对类C依赖的实现。对于类B和类D来说，虽然他们都存在着用不到的方法（也就是图中红色字体标记的方法），但由于实现了接口I，所以也必须要实现这些用不到的方法。对类图不熟悉的可以参照程序代码来理解，代码如下：

```
interface I {
```

```
    public void method1();
    public void method2();
    public void method3();
    public void method4();
    public void method5();
}

class A{
    public void depend1(I i){
        i.method1();
    }
    public void depend2(I i){
        i.method2();
    }
    public void depend3(I i){
        i.method3();
    }
}

class B implements I{
    public void method1() {
        System.out.println("类B实现接口I的方法1");
    }
    public void method2() {
        System.out.println("类B实现接口I的方法2");
    }
    public void method3() {
        System.out.println("类B实现接口I的方法3");
    }
    //对于类B来说，method4和method5不是必需的，但是由于接口A中有这两个方法，
    //所以在实现过程中即使这两个方法的方法体为空，也要将这两个没有作用的方法进
    public void method4() {}
    public void method5() {}
}

class C{
    public void depend1(I i){
        i.method1();
    }
    public void depend2(I i){
        i.method4();
    }
    public void depend3(I i){
        i.method5();
    }
}

class D implements I{
    public void method1() {
        System.out.println("类D实现接口I的方法1");
    }
    //对于类D来说，method2和method3不是必需的，但是由于接口A中有这两个方法，
    //所以在实现过程中即使这两个方法的方法体为空，也要将这两个没有作用的方法进
```

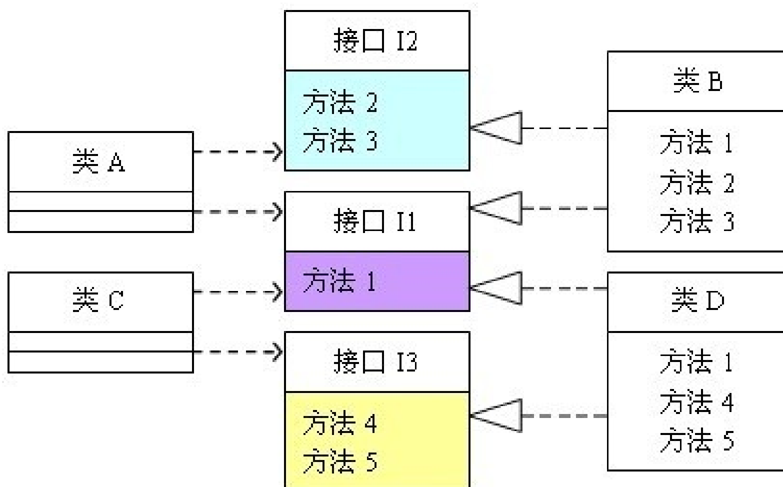
```
    public void method2() {}
    public void method3() {}

    public void method4() {
        System.out.println("类D实现接口I的方法4");
    }
    public void method5() {
        System.out.println("类D实现接口I的方法5");
    }
}

public class Client{
    public static void main(String[] args){
        A a = new A();
        a.depend1(new B());
        a.depend2(new B());
        a.depend3(new B());

        C c = new C();
        c.depend1(new D());
        c.depend2(new D());
        c.depend3(new D());
    }
}
```

可以看到，如果接口过于臃肿，只要接口中出现的方法，不管对依赖于它的类有没有用处，实现类中都必须去实现这些方法，这显然不是好的设计。如果将这个设计修改为符合接口隔离原则，就必须对接口I进行拆分。在这里我们将原有的接口I拆分为三个接口，拆分后的设计如图2所示：




```
interface I1 {
    public void method1();
}

interface I2 {
```

```
    public void method2();
    public void method3();
}

interface I3 {
    public void method4();
    public void method5();
}

class A{
    public void depend1(I1 i){
        i.method1();
    }
    public void depend2(I2 i){
        i.method2();
    }
    public void depend3(I2 i){
        i.method3();
    }
}

class B implements I1, I2{
    public void method1() {
        System.out.println("类B实现接口I1的方法1");
    }
    public void method2() {
        System.out.println("类B实现接口I2的方法2");
    }
    public void method3() {
        System.out.println("类B实现接口I2的方法3");
    }
}

class C{
    public void depend1(I1 i){
        i.method1();
    }
    public void depend2(I3 i){
        i.method4();
    }
    public void depend3(I3 i){
        i.method5();
    }
}

class D implements I1, I3{
    public void method1() {
        System.out.println("类D实现接口I1的方法1");
    }
    public void method4() {
        System.out.println("类D实现接口I3的方法4");
    }
    public void method5() {
```

```
            System.out.println("类D实现接口I3的方法5");
        }
    }
```

接口隔离原则的含义是：建立单一接口，不要建立庞大臃肿的接口，尽量细化接口，接口中的方法尽量少。也就是说，我们要为各个类建立专用的接口，而不要试图去建立一个很庞大的接口供所有依赖它的类去调用。本文例子中，将一个庞大的接口变更为3个专用的接口所采用的就是接口隔离原则。在程序设计中，依赖几个专用的接口要比依赖一个综合的接口更灵活。接口是设计时对外部设定的“契约”，通过分散定义多个接口，可以预防外来变更的扩散，提高系统的灵活性和可维护性。

说到这里，很多人会觉的接口隔离原则跟之前的单一职责原则很相似，其实不然。其一，单一职责原则原注重的是职责；而接口隔离原则注重对接口依赖的隔离。其二，单一职责原则主要是约束类，其次才是接口和方法，它针对的是程序中的实现和细节；而接口隔离原则主要约束接口接口，主要针对抽象，针对程序整体框架的构。 采用接口隔离原则对接口进行约束时，要注意以下几点：

- 接口尽量小，但是要有限度。对接口进行细化可以提高程序设计灵活性是不挣的事实，但是如果过小，则会造成接口数量过多，使设计复杂化。所以一定要适度。
- 为依赖接口的类定制服务，只暴露给调用的类它需要的方法，它不需要的方法则隐藏起来。只有专注地为一个模块提供定制服务，才能建立最小的依赖关系。
- 提高内聚，减少对外交互。使接口用最少的方法去完成最多的事情。 运用接口隔离原则，一定要适度，接口设计的过大或过小都不好。设计接口的时候，只有多花些时间去思考和筹划，才能准确地实践这一原则。

迪米特原则(Law of Demeter)

简介

得墨忒耳定律（Law of Demeter，缩写LoD）亦稱爲“最少知识原则（Principle of Least Knowledge）”，是一种软件开发的设计指導原则，特别是面向对象的程序设计。得墨忒耳定律是松耦合的一种具體案例。該原則是美國東北大學在1987年末在發明的，可以簡單地以下面任一種方式總結:

每个单元对于其他的单元只能拥有有限的知识：只是与当前单元紧密联系的单元；每个单元只能和它的朋友交谈：不能和陌生单元交谈；只和自己直接的朋友交谈。这个原理的名称来源于希腊神话中的农业女神，孤独的得墨忒耳。

很多面向对象程序设计语言用"."表示对象的域的解析算符，因此得墨忒耳定律可以简单地陈述为“只使用一个.算符”。因此，a.b.Method()违反了此定律，而a.Method()不违反此定律。一个简单例子是，人可以命令一条狗行走（walk），但是不应该直接指挥狗的腿行走，应该由狗去指挥控制它的腿如何行走。

实例

定义：一个对象应该对其他对象保持最少的了解。 问题由来：类与类之间的关系越密切，耦合度越大，当一个类发生改变时，对另一个类的影响也越大。

解决方案：尽量降低类与类之间的耦合。

自从我们接触编程开始，就知道了软件编程的总的原则：低耦合，高内聚。无论是面向过程编程还是面向对象编程，只有使各个模块之间的耦合尽量的低，才能提高代码的复用率。低耦合的优点不言而喻，但是怎么样编程才能做到低耦合呢？那正是迪米特法则要去完成的。

迪米特法则又叫最少知道原则，最早是在1987年由美国Northeastern University的Ian Holland提出。通俗的来讲，就是一个类对自己依赖的类知道的越少越好。也就是说，对于被依赖的类来说，无论逻辑多么复杂，都尽量地的将逻辑封装在类的内部，对外除了提供的public方法，不对外泄漏任何信息。迪米特法则还有一个更简单的定义：只与直接的朋友通信。首先来解释一下什么是直接的朋友：每个对象都会与其他对象有耦合关系，只要两个对象之间有耦合关系，我们就说这两个对象之间是朋友关系。耦合的方式很多，依赖、关联、组合、聚合等。其中，我们称出现成员变量、方法参数、方法返回值中的类为直接的朋友，而出现在局部变量中的类则不是直接的朋友。也就是说，陌生的类最好不要作为局部变量的形式出现在类的内部。

举一个例子：有一个集团公司，下属单位有分公司和直属部门，现在要求打印出所有下属单位的员工ID。先来看一下违反迪米特法则的设计。

```
//总公司员工
```

```
class Employee{
    private String id;
    public void setId(String id){
        this.id = id;
    }
    public String getId(){
        return id;
    }
}

//分公司员工
class SubEmployee{
    private String id;
    public void setId(String id){
        this.id = id;
    }
    public String getId(){
        return id;
    }
}

class SubCompanyManager{
    public List<SubEmployee> getAllEmployee(){
        List<SubEmployee> list = new ArrayList<SubEmployee>();
        for(int i=0; i<100; i++){
            SubEmployee emp = new SubEmployee();
            //为分公司人员按顺序分配一个ID
            emp.setId("分公司"+i);
            list.add(emp);
        }
        return list;
    }
}

class CompanyManager{

    public List<Employee> getAllEmployee(){
        List<Employee> list = new ArrayList<Employee>();
        for(int i=0; i<30; i++){
            Employee emp = new Employee();
            //为总公司人员按顺序分配一个ID
            emp.setId("总公司"+i);
            list.add(emp);
        }
        return list;
    }

    public void printAllEmployee(SubCompanyManager sub){
        List<SubEmployee> list1 = sub.getAllEmployee();
        for(SubEmployee e:list1){
            System.out.println(e.getId());
        }
```

```
        List<Employee> list2 = this.getAllEmployee();
        for(Employee e:list2){
            System.out.println(e.getId());
        }
    }
}

public class Client{
    public static void main(String[] args){
        CompanyManager e = new CompanyManager();
        e.printAllEmployee(new SubCompanyManager());
    }
}
```

现在这个设计的主要问题出在CompanyManager中，根据迪米特法则，只与直接的朋友发生通信，而SubEmployee类并不是CompanyManager类的直接朋友（以局部变量出现的耦合不属于直接朋友），从逻辑上讲总公司只与他的分公司耦合就行了，与分公司的员工并没有任何联系，这样设计显然是增加了不必要的耦合。按照迪米特法则，应该避免类中出现这样非直接朋友关系的耦合。修改后的代码如下:

```
class SubCompanyManager{
    public List<SubEmployee> getAllEmployee(){
        List<SubEmployee> list = new ArrayList<SubEmployee>();
        for(int i=0; i<100; i++){
            SubEmployee emp = new SubEmployee();
            //为分公司人员按顺序分配一个ID
            emp.setId("分公司"+i);
            list.add(emp);
        }
        return list;
    }
    public void printEmployee(){
        List<SubEmployee> list = this.getAllEmployee();
        for(SubEmployee e:list){
            System.out.println(e.getId());
        }
    }
}

class CompanyManager{
    public List<Employee> getAllEmployee(){
        List<Employee> list = new ArrayList<Employee>();
        for(int i=0; i<30; i++){
            Employee emp = new Employee();
            //为总公司人员按顺序分配一个ID
            emp.setId("总公司"+i);
            list.add(emp);
        }
        return list;
    }

    public void printAllEmployee(SubCompanyManager sub){
        sub.printEmployee();
        List<Employee> list2 = this.getAllEmployee();
        for(Employee e:list2){
            System.out.println(e.getId());
        }
    }
}
```

修改后，为分公司增加了打印人员ID的方法，总公司直接调用来打印，从而避免了与分公司的员工发生耦合。

迪米特法则的初衷是降低类之间的耦合，由于每个类都减少了不必要的依赖，因此的确可以降低耦合关系。但是凡事都有度，虽然可以避免与非直接的类通信，但是要通信，必然会通过一个“中介”来发生联系，例如本例中，总公司就是通过分公司这个“中介”来与分公司的员工发生联系的。过分的使用迪米特原则，会产生大量这样的中介和传递类，导致系统复杂度变大。所以在采用迪米特法则时要反复权衡，既做到结构清晰，又要高内聚低耦合。

开闭原则(Open-Close Principle)

简介

Software entities (classes, modules, functions, etc.) should be open for extension but closed for modification. 软件实体（类，模块，方法等等）应当对扩展开放，对修改关闭，即软件实体应当在不修改的前提下扩展。

问题由来:在软件的生命周期内，因为变化、升级和维护等原因需要对软件原有代码进行修改时，可能会给旧代码中引入错误，也可能会使我们不得不对整个功能进行重构，并且需要原有代码经过重新测试。

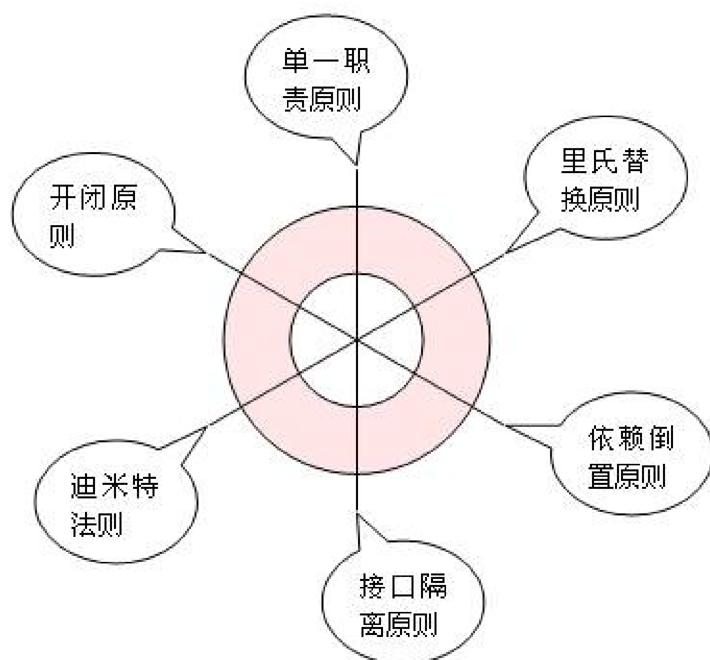
解决方案: 开闭原则是面向对象设计中最基础的设计原则，它指导我们如何建立稳定灵活的系统。开闭原则可能是设计模式六项原则中定义最模糊的一个了，它只告诉我们对扩展开放，对修改关闭，可是到底如何才能做到对扩展开放，对修改关闭，并没有明确的告诉我们。以前，如果有人告诉我“你进行设计的时候一定要遵守开闭原则”，我会觉的他什么都没说，但貌似又什么都说了。因为开闭原则真的太虚了。

在仔细思考以及仔细阅读很多设计模式的文章后，终于对开闭原则有了一点认识。其实，我们遵循设计模式其它5大原则，以及使用23种设计模式的目的就是遵循开闭原则。也就是说，只要我们对其它5项原则遵守的好了，设计出的软件自然是符合开闭原则的，这个开闭原则更像是前面五项原则遵守程度的“平均得分”，前面5项原则遵守的好，平均分自然就高，说明软件设计开闭原则遵守的好；如果前面5项原则遵守的不好，则说明开闭原则遵守的不好。

开闭原则无非就是想表达这样一层意思：用抽象构建框架，用实现扩展细节。因为抽象灵活性好，适应性广，只要抽象的合理，可以基本保持软件架构的稳定。而软件中易变的细节，我们用从抽象派生的实现类来进行扩展，当软件需要发生变化时，我们只需要根据需求重新派生一个实现类来扩展就可以了。当然前提是我们的抽象要合理，要对需求的变更有前瞻性和预见性才行。

其它的**5**项原则，恰恰是告诉我们用抽象构建框架，用实现扩展细节的注意事项而已：单一职责原则告诉我们实现类要职责单一；里氏替换原则告诉我们不要破坏继承体系；依赖倒置原则告诉我们要面向接口编程；接口隔离原则告诉我们在设计接口的时候要精简单一；迪米特法则告诉我们要降低耦合。而开闭原则是总纲，他告诉我们要对扩展开放，对修改关闭。

最后说明一下如何去遵守这六个原则。对这六个原则的遵守并不是是和否的问题，而是多和少的问题，也就是说，我们一般不会说有没有遵守，而是说遵守程度的多少。任何事都是过犹不及，设计模式的六个设计原则也是一样，制定这六个原则的目的并不是要我们刻板的遵守他们，而需要根据实际情况灵活运用。对他们的遵守程度只要在一个合理的范围内，就算是良好的设计。我们用一幅图来说明一下。



图中的每一条维度各代表一项原则，我们依据对这项原则的遵守程度在维度上画一个点，则如果对这项原则遵守的合理的话，这个点应该落在红色的同心圆内部；如果遵守的差，点将会在小圆内部；如果过度遵守，点将会落在大圆外部。一个良好的设计体现在图中，应该是六个顶点都在同心圆中的六边形。

实例

在软件开发过程中，永远不变的就是变化。开闭原则是使我们的软件系统拥抱变化的核心原则之一。对扩展可放，对修改关闭给出了高层次的概括，即在需要对软件进行升级、变化时应该通过扩展的形式来实现，而非修改原有代码。当然这只是一种比较理想的状态，是通过扩展还是通过修改旧代码需要根据代码自身来定。

在Volley中，开闭原则体现得比较好的是Request类族的设计。我们知道，在开发C/S应用时，服务器返回的数据格式多种多样，有字符串类型、xml、json等。而解析服务器返回的Response的原始数据类型则是通过Request类来实现的，这样就使得Request类对于服务器返回的数据格式有良好的扩展性，即Request的可变性太大。

例如我们返回的数据格式是Json，那么我们使用JsonObjectRequest请求来获取数据，它会将结果转成JsonObject对象，我们看看JsonObjectRequest的核心实现。

```
/**
 * A request for retrieving a {@link JSONObject} response body at a
 * optional {@link JSONObject} to be passed in as part of the reque
 */
public class JsonObjectRequest extends JsonRequest<JSONObject> {
   // 代码省略
    @Override
    protected Response<JSONObject> parseNetworkResponse(NetworkResp
        try {
            String jsonString =
                new String(response.data, HttpHeaderParser.parseCha
            return Response.success(new JSONObject(jsonString),
                    HttpHeaderParser.parseCacheHeaders(response));
        } catch (UnsupportedEncodingException e) {
            return Response.error(new ParseError(e));
        } catch (JSONException je) {
            return Response.error(new ParseError(je));
        }
    }
}
```

JsonObjectRequest通过实现Request抽象类的parseNetworkResponse解析服务器返回的结果，这里将结果转换为JSONObject，并且封装到Response类中。

例如Volley添加对图片请求的支持，即ImageLoader(已内置)。这个时候我的请求返回的数据是Bitmap图片。因此我需要在该类型的Request得到的结果是Request，但支持一种数据格式不能通过修改源码的形式，这样可能会为旧代码引入错误。但是你又需要支持新的数据格式，此时我们的开闭原则就很重要了，对扩展开放，对修改关闭。我们看看Volley是如何做的。

```
/**
 * A canned request for getting an image at a given URL and calling
 * back with a decoded Bitmap.
 */
public class ImageRequest extends Request<Bitmap> {
    // 代码省略
    // 将结果解析成Bitmap，并且封装套Response对象中
    @Override
    protected Response<Bitmap> parseNetworkResponse(NetworkResponse
        // Serialize all decode on a global lock to reduce concurre
        synchronized (sDecodeLock) {
            try {
                return doParse(response);
            } catch (OutOfMemoryError e) {
                VolleyLog.e("Caught OOM for %d byte image, url=%s",
                return Response.error(new ParseError(e));
            }
        }
    }
    /**
     * The real guts of parseNetworkResponse. Broken out for readab
     */
    private Response<Bitmap> doParse(NetworkResponse response) {
        byte[] data = response.data;
        BitmapFactory.Options decodeOptions = new BitmapFactory.Opt
        Bitmap bitmap = null;
        // 解析Bitmap的相关代码,在此省略
        if (bitmap == null) {
            return Response.error(new ParseError(response));
        } else {
            return Response.success(bitmap, HttpHeaderParser.parseC
        }
    }
}
```

需要添加某种数据格式的Request时，只需要继承自Request类，并且实现相应的方法即可。这样通过扩展的形式来应对软件的变化或者说用户需求的多样性，即避免了破坏原有系统，又保证了软件系统的可扩展性。

装饰原则(Decorator pattern)

简介

Decorator装饰模式是一种结构型模式，它主要是解决：“过度地使用了继承来扩展对象的功能”，由于继承为类型引入的静态特质，使得这种扩展方式缺乏灵活性；并且随着子类的增多（扩展功能的增多），各种子类的组合（扩展功能的组合）会导致更多子类的膨胀（多继承）。继承为类型引入的静态特质的意思是说以继承的方式使某一类型要获得功能是在编译时。所谓静态，是指在编译时；动态，是指在运行时。

修饰模式，是面向对象编程领域中，一种动态地往一个类中添加新的行为的设计模式。就功能而言，修饰模式相比生成子类更为灵活，这样可以给某个对象而不是整个类添加一些功能。

GoF《设计模式》中说道：动态的给一个对象添加一些额外的职责。就增加功能而言，Decorator模式比生成子类更为灵活。 下面来看看Decorator模式的结构：



看这个结构好像不是很明白，下面我根据代码讲解一下这个结构。我想了一个场景：我们现在用的手机功能很多，我就用Decorator模式实现一下对某个手机的GSP和蓝牙功能扩展。 首先，我们需要一个手机的接口或者是抽象类，我这里就用抽象类来实现，代码如下：

```
public abstract class AbstractCellPhone
    {
        public abstract string CallNumber();
        public abstract string SendMessage();
}
```

AbstractCellPhone也就是结构图中的Component，然后，我再来实现Nokia和Moto的手机类，这类要继承AbstractCellPhone，也就是图中ConcreteComponent类要继承Component，实现代码如下：

```
public class NokiaPhone : AbstractCellPhone
    {
        public override string CallNumber()
        {
            return "NokiaPhone call sombody";
        }

        public override string SendMessage()
        {
            return "NokiaPhone send a message to somebody";
        }
    }

    public class MotoPhone : AbstractCellPhone
    {
        public override string CallNumber()
        {
            return "MotoPhone call sombody";
        }

        public override string SendMessage()
        {
            return "MotoPhone send a message to somebody";
        }
 }
```

接下来我需要一个Decorator接口或者抽象类，实现代码如下：

```
public abstract class Decorator:AbstractCellPhone
    {
        AbstractCellPhone _phone;

        public Decorator(AbstractCellPhone phone)
        {
            _phone = phone;
        }

        public override string CallNumber()
        {
            return _phone.CallNumber();
        }

        public override string SendMessage()
        {
            return _phone.SendMessage();
        }
  }
```

正如结构图中，这个Decorator即继承了AbstractCellPhone，又包含了一个私有的AbstractCellPhone的对象。这样做的意义是：Decorator类又使用了另外一个Component类。我们可以使用一个或多个Decorator对象来“装饰”一个Component对象，且装饰后的对象仍然是一个Component对象。在下来，我要实现GSP和蓝牙的功能扩展，它们要继承自Decorator，代码如下：

```
public class DecoratorGPS : Decorator
    {
        public DecoratorGPS(AbstractCellPhone phone)
            : base(phone)
        { }

        public override string CallNumber()
        {
            return base.CallNumber() + " with GPS";
        }

        public override string SendMessage()
        {
            return base.SendMessage() + " with GPS";
        }
    }

    public class DecoratorBlueTooth : Decorator
    {
        public DecoratorBlueTooth(AbstractCellPhone phone)
            : base(phone)
        { }

        public override string CallNumber()
        {
            return base.CallNumber() + " with BlueTooth";
        }

        public override string SendMessage()
        {
            return base.SendMessage() + " with BlueTooth";
        }
 }
```

最后，用客户端程序验证一下：

```
static void Main(string[] args)
        {
             AbstractCellPhone phone = new NokiaPhone();
            Console.WriteLine(phone.CallNumber());
            Console.WriteLine(phone.SendMessage());
            DecoratorGPS gps = new DecoratorGPS(phone);     //add G
            Console.WriteLine(gps.CallNumber());
            Console.WriteLine(gps.SendMessage());
            DecoratorBlueTooth bluetooth = new DecoratorBlueTooth(g
            Console.WriteLine(bluetooth.CallNumber());
            Console.WriteLine(bluetooth.SendMessage());
            Console.Read();
     }
```

执行结果： NokiaPhone call sombody NokiaPhone send a message to somebody NokiaPhone call sombody with GPS NokiaPhone send a message to somebody with GPS NokiaPhone call sombody with GPS with BlueTooth NokiaPhone send a message to somebody with GPS with BlueTooth

从执行的结果不难看出扩展功能已被添加。最后再说说Decorator装饰模式的几点要点：

1. 通过采用组合、而非继承的手法，Decorator模式实现了在运行时动态的扩展对象功能的能力，而且可以根据需要扩展多个功能。避免了单独使用继承带来的“灵活性差”和“多子类衍生问题”。
2. Component类在Decorator模式中充当抽象接口的角色，不应该去实现具体的行为。而且Decorator类对于Component类应该透明——换言之Component类无需知道Decorator类，Decorator类是从外部来扩展Component类的功能。
3. Decorator类在接口上表现为is-a Component的继承关系，即Decorator类继承了Component类所具有的接口。但在实现上又表现为has-a Component的组合关系，即Decorator类又使用了另外一个Component类。我们可以使用一个或多个Decorator对象来“装饰”一个Component对象，且装饰后的对象仍然是一个Component对象。（在这里我想谈一下我的理解：当我们实例化一个Component对象后，要给这个对象扩展功能，这时我们把这个Component对象当作参数传给Decorator的子类的构造函数——也就是扩展方法的功能类。对于引用类型传参时，实际上只是传递对象的地址，这样，在功能扩展是，操作的应该是同一个对象）
4. Decorator模式并非解决“多子类衍生的多继承”问题，Decorator模式应用的要点在于解决“主体类在多个方向上的扩展功能”——是为“装饰”的含义。Decorator是在运行时对功能进行组合。

实例

Category

Objective-C 中的 Category 就是对装饰模式的一种具体实现。它的主要作用是在不改变原有类的前提下，动态地给这个类添加一些方法。在 Objective-C 中的具体体现为：实例（类）方法、属性和协议。是的，在 Objective-C 中可以用 Category 来实现协议。本文将结合 runtime（我下载的是当前的最新版本 objc4-646.tar.gz) 的源码来探究它实现的原理。

使用场景

根据苹果官方文档对 Category 的描述，它的使用场景主要有三个：

- 给现有的类添加方法.
- 将一个类的实现拆分成多个独立的源文件.
- 声明私有的方法.

  其中，第 1 个是最典型的使用场景，应用最广泛。

注：Category 有一个非常容易误用的场景，那就是用 Category 来覆写父类或主类的方法。虽然目前 Objective-C 是允许这么做的，但是这种使用场景是非常不推荐的。使用 Category 来覆写方法有很多缺点，比如不能覆写 Category 中的方法、无法调用主类中的原始实现等，且很容易造成无法预估的行为。

我们知道，无论我们有没有主动引入 Category 的头文件，Category 中的方法都会被添加进主类中。我们可以通过 - performSelector: 等方式对 Category 中的相应方法进行调用，之所以需要在调用的地方引入 Category 的头文件，只是为了“照顾”编译器同学的感受。

下面，我们将结合 runtime 的源码探究下 Category 的实现原理。打开 runtime 源码工程，在文件 objc-runtime-new.mm 中找到以下函数：

```
void _read_images(header_info **hList, uint32_t hCount)
{
    ...
        _free_internal(resolvedFutureClasses);
    }

    // Discover categories.
    for (EACH_HEADER) {
        category_t **catlist =
            _getObjc2CategoryList(hi, &count);
        for (i = 0; i < count; i++) {
            category_t *cat = catlist[i];
            Class cls = remapClass(cat->cls);

            if (!cls) {
                // Category's target class is missing (probably wea
                // Disavow any knowledge of this category.
                catlist[i] = nil;
                if (PrintConnecting) {
                    _objc_inform("CLASS: IGNORING category \?\?\?(%
                                 "missing weak-linked target class'
                                 cat->name, cat);
```

```
                }
                continue;
            }

              ***
            // Process this category.
            // First, register the category with its target class.
            // Then, rebuild the class's method lists (etc) if
            // the class is realized.
            BOOL classExists = NO;
            if (cat->instanceMethods ||  cat->protocols
                ||  cat->instanceProperties)
            {
                addUnattachedCategoryForClass(cat, cls, hi);
                if (cls->isRealized()) {
                    remethodizeClass(cls);
                    classExists = YES;
                }
                if (PrintConnecting) {
                    _objc_inform("CLASS: found category -%s(%s) %s'
                                 cls->nameForLogging(), cat->name,
                                 classExists ? "on existing class"
                }
            }

            if (cat->classMethods  ||  cat->protocols
                /* ||  cat->classProperties */)
            {
                addUnattachedCategoryForClass(cat, cls->ISA(), hi);
                if (cls->ISA()->isRealized()) {
                    remethodizeClass(cls->ISA());
                }
                if (PrintConnecting) {
                    _objc_inform("CLASS: found category +%s(%s)",
                                 cls->nameForLogging(), cat->name);
                }
            }
        }
    }

    // Category discovery MUST BE LAST to avoid potential races
    // when other threads call the new category code before
    // this thread finishes its fixups.

    // +load handled by prepare_load_methods()

    ...
}
```

从第 27-58 行的关键代码，我们可以知道在这个函数中对 Category 做了如下处理：

- 将 Category 和它的主类（或元类）注册到哈希表中；
- 如果主类（或元类）已实现，那么重建它的方法列表。 在这里分了两种情况进行处理：Category 中的实例方法和属性被整合到主类中；而类方法则被整合到元类中（关于对象、类和元类的更多细节，可以参考博文Objective-C 对象模型。另外，对协议的处理比较特殊，Category 中的协议被同时整合到了主类和元类中。

我们注意到，不管是哪种情况，最终都是通过调用 static void remethodizeClass(Class cls) 函数来重新整理类的数据的。

```
static void remethodizeClass(Class cls)
{
    ...
                             cls->nameForLogging(), isMeta ? "(meta)"
        }

        // Update methods, properties, protocols

        attachCategoryMethods(cls, cats, YES);

        newproperties = buildPropertyList(nil, cats, isMeta);
        if (newproperties) {
            newproperties->next = cls->data()->properties;
            cls->data()->properties = newproperties;
        }

        newprotos = buildProtocolList(cats, nil, cls->data()->proto
        if (cls->data()->protocols  &&  cls->data()->protocols != n
            _free_internal(cls->data()->protocols);
        }
        cls->data()->protocols = newprotos;

        _free_internal(cats);
    }
}
```

这个函数的主要作用是将 Category 中的方法、属性和协议整合到类（主类或元类）中，更新类的数据字段 `data()` 中 `method_lists` （或 `method_list` ）、 `properties` 和 `protocols` 的值。进一步，我们通过 `attachCategoryMethods` 函数的源码可以找到真正处理 Category 方法的 `attachMethodLists` 函数：

```
static void
attachMethodLists(Class cls, method_list_t **addedLists, int addedC
                  bool baseMethods, bool methodsFromBundle,
                  bool flushCaches)
{
    ...
        newLists[newCount++] = mlist;
    }

    // Copy old methods to the method list array
    for (i = 0; i < oldCount; i++) {
        newLists[newCount++] = oldLists[i];
    }
    if (oldLists  &&  oldLists != oldBuf) free(oldLists);

    // nil-terminate
    newLists[newCount] = nil;

    if (newCount > 1) {
        assert(newLists != newBuf);
        cls->data()->method_lists = newLists;
        cls->setInfo(RW_METHOD_ARRAY);
    } else {
        assert(newLists == newBuf);
        cls->data()->method_list = newLists[0];
        assert(!(cls->data()->flags & RW_METHOD_ARRAY));
    }
}
```

这个函数的代码量看上去比较多，但是我们并不难理解它的目的。它的主要作用就是将类中的旧有方法和 Category 中新添加的方法整合成一个新的方法列表，并赋值给 method_lists 或 method_list 。通过探究这个处理过程，我们也印证了一个结论，那就是主类中的方法和 Category 中的方法在 runtime 看来并没有区别，它们是被同等对待的，都保存在主类的方法列表中。

不过，类的方法列表字段有一点特殊，它的结构是联合体，method_lists 和 method_list 共用同一块内存地址。当 newCount 的个数大于 1 时，使用 method_lists 来保存 newLists ，并将方法列表的标志位置为 RW_METHOD_ARRAY ，此时类的方法列表字段是 method_list_t 类型的指针数组；否则，使用 method_list 来保存 newLists ，并将方法列表的标志位置空，此时类的方法列表字段是 method_list_t 类型的指针。

```
// class's method list is an array of method lists
#define RW_METHOD_ARRAY       (1<<20)

union {
    method_list_t **method_lists;  // RW_METHOD_ARRAY == 1
    method_list_t *method_list;    // RW_METHOD_ARRAY == 0
};
```

我们注意到 runtime 对 Category 中方法的处理过程并没有对 +load 方法进行什么特殊地处理。因此，严格意义上讲 Category 中的 +load 方法跟普通方法一样也会对主类中的 +load 方法造成覆盖，只不过 runtime 在自动调用主类和 Category 中的 +load 方法时是直接使用各自方法的指针进行调用的。所以才会使我们觉得主类和 Category 中的 +load 方法好像互不影响一样。因此，当我们手动给主类发送 +load 消息时，调用的一直会是分类中的 +load 方法，you should give it a try yourself 。

代理模式(Proxy pattern)

简介

代理模式（英语：Proxy Pattern）是程序设计中的一种设计模式。

所谓的代理者是指一个类可以作为其它东西的接口。代理者可以作任何东西的接口：网络连接、内存中的大对象、文件或其它昂贵或无法复制的资源。

著名的代理模式例子为引用计数（英语：reference counting）指针对象。

当一个复杂对象的多份副本须存在时，代理模式可以结合享元模式以减少内存用量。典型作法是创建一个复杂对象及多个代理者，每个代理者会引用到原本的复杂对象。而作用在代理者的运算会转送到原本对象。一旦所有的代理者都不存在时，复杂对象会被移除。

在某些情况下，一个客户不想或者不能直接引用一个对象，此时可以通过一个称之为“代理”的第三者来实现 间接引用。代理对象可以在客户端和目标对象之间起到 中介的作用，并且可以通过代理对象去掉客户不能看到 的内容和服务或者添加客户需要的额外服务。

通过引入一个新的对象（如小图片和远程代理 对象）来实现对真实对象的操作或者将新的对 象作为真实对象的一个替身，这种实现机制即 为代理模式，通过引入代理对象来间接访问一 个对象，这就是代理模式的模式动机。

讲解

```
#include <iostream>
#include "RealSubject.h"
#include "Proxy.h"

using namespace std;

int main(int argc, char *argv[])
{
    Proxy proxy;
    proxy.request();

    return 0;
}
```

```
///////////////////////////////////////////////////////////
//  Proxy.h
//  Implementation of the Class Proxy
//  Created on:      07-十月-2014 16:57:54
//  Original author: colin
///////////////////////////////////////////////////////////

#if !defined(EA_56011290_0413_40c6_9132_63EE89B023FD__INCLUDED_)
#define EA_56011290_0413_40c6_9132_63EE89B023FD__INCLUDED_

#include "RealSubject.h"
#include "Subject.h"

class Proxy : public Subject
{

public:
    Proxy();
    virtual ~Proxy();

    void request();

private:
    void afterRequest();
    void preRequest();
    RealSubject *m_pRealSubject;

};
#endif // !defined(EA_56011290_0413_40c6_9132_63EE89B023FD__INCLUDE
```

```
////////////////////////////////////////////////////////////
//  Proxy.cpp
//  Implementation of the Class Proxy
//  Created on:      07-十月-2014 16:57:54
//  Original author: colin
////////////////////////////////////////////////////////////

#include "Proxy.h"
#include <iostream>
using namespace std;


Proxy::Proxy(){
    //有人觉得 RealSubject对象的创建应该是在main中实现；我认为RealSubject
    //对用户是透明的，用户所面对的接口都是通过代理的；这样才是真正的代理；
    m_pRealSubject = new RealSubject();
}

Proxy::~Proxy(){
    delete m_pRealSubject;
}

void Proxy::afterRequest(){
    cout << "Proxy::afterRequest" << endl;
}


void Proxy::preRequest(){
    cout << "Proxy::preRequest" << endl;
}


void Proxy::request(){
    preRequest();
    m_pRealSubject->request();
    afterRequest();
}
```

优点

- 代理模式能够协调调用者和被调用者，在一定程度上降低了系 统的耦合度。
- 远程代理使得客户端可以访问在远程机器上的对象，远程机器 可能具有更好的计算性能与处理速度，可以快速响应并处理客户端请求。
- 虚拟代理通过使用一个小对象来代表一个大对象，可以减少系 统资源的消耗，对系统进行优化并提高运行速度。
- 保护代理可以控制对真实对象的使用权限。

缺点

- 由于在客户端和真实主题之间增加了代理对象，因此 有些类型的代理模式可能会造成请求的处理速度变慢。
- 实现代理模式需要额外的工作，有些代理模式的实现 非常复杂。

适用环境

根据代理模式的使用目的，常见的代理模式有以下几种类型：

- 远程(Remote)代理：为一个位于不同的地址空间的对象提供一个本地 的代理对象，这个不同的地址空间可以是在同一台主机中，也可是在 另一台主机中，远程代理又叫做大使(Ambassador)。
- 虚拟(Virtual)代理：如果需要创建一个资源消耗较大的对象，先创建一个消耗相对较小的对象来表示，真实对象只在需要时才会被真正创建。
- Copy-on-Write代理：它是虚拟代理的一种，把复制（克隆）操作延迟 到只有在客户端真正需要时才执行。一般来说，对象的深克隆是一个 开销较大的操作，Copy-on-Write代理可以让这个操作延迟，只有对象被用到的时候才被克隆。
- 保护(Protect or Access)代理：控制对一个对象的访问，可以给不同的用户提供不同级别的使用权限。
- 缓冲(Cache)代理：为某一个目标操作的结果提供临时的存储空间，以便多个客户端可以共享这些结果。
- 防火墙(Firewall)代理：保护目标不让恶意用户接近。
- 同步化(Synchronization)代理：使几个用户能够同时使用一个对象而没有冲突。
- 智能引用(Smart Reference)代理：当一个对象被引用时，提供一些额外的操作，如将此对象被调用的次数记录下来等。

实例

图片懒加载是一个很典型的代理模式的例子。

```
//抽象日志记录类：抽象主题
interface AbstractLog
{
    public void method();
}
```

```
import java.util.*;

//日志记录代理类：代理主题
class LoggerProxy implements AbstractLog
{
    private BusinessClass business;

    public LoggerProxy()
    {
        business = new BusinessClass();
    }

    public void method()
    {
        Calendar calendar = new GregorianCalendar();
        int year = calendar.get(Calendar.YEAR);
        int month = calendar.get(Calendar.MONTH) + 1;
        int day = calendar.get(Calendar.DAY_OF_MONTH);
        int hour = calendar.get(Calendar.HOUR) + 12;
        int minute = calendar.get(Calendar.MINUTE);
        int second = calendar.get(Calendar.SECOND);
        String dateTime = year + "-" + month + "-" + day + " " + ho
        System.out.println("方法method()被调用，调用时间为" + dateTime
        try{
            business.method();
            System.out.println("方法method()调用成功！");
        }
        catch(Exception e)
        {
            System.out.println("方法method()调用失败！");
        }
    }
}
```

```
//业务类：真实主题
class BusinessClass implements AbstractLog
{
    public void method()
    {
        System.out.println("真实业务方法！");
    }
}
```

```
//客户端测试类
class Client
{
    public static void main(String args[])
    {
        AbstractLog al;
        al = new LoggerProxy();
        al.method();
    }
}
```

在本实例中，通过代理类LoggerProxy来间接调用真实业务类BusinessClass的方法，可以在调用真实业务方法时增加新功能（如日志记录），此处使用的是代理模式的一种较为简单的形式，类似于保护代理，在实施真实调用时可以执行一些额外的操作。由于代理主题和真实主题实现了相同的接口，因此在客户端可以针对抽象编程，而将具体代理类类名存储至配置文件中，增加和更换代理类和真实类都很方便，无需修改源代码，满足开闭原则。

工厂方法模式(Factory Method pattern)

简介

“工厂方法模式(Factory Method Pattern)又称为工厂模式，也叫虚拟构造器(Virtual Constructor)模式或者多态工厂(Polymorphic Factory)模式，它属于类创建型模式。在工厂方法模式中，工厂父类负责定义创建产品对象的公共接口，而工厂子类则负责生成具体的产品对象，这样做的目的是将产品类的实例化操作延迟到工厂子类中完成，即通过工厂子类来确定究竟应该实例化哪一个具体产品类。

与简单工厂模式的对比

以计算器为例

简单工厂模式

```
public class OperationFactory
{
    public static operation createOperate(string operate)
    {
    Operation oper = null;
    switch (operate)
    {
        case "+":
            oper = new OperationAdd();
            break;
        case "-":
            oper = new OperationSub();
            break;
        case "*":
            oper = new OperationMul();
            break;
        case "/":
            oper = new OperationDiv();
            break;
    }
    return oper;
    }
}
```

客户端代码

```
Operation oper;
oper = OperationFactory.createOperate("+");
oper.NumberA = 1;
oper.NumberB = 2;
double result = oper.GetResult();
```

工厂方法模式

```
interface IFactory
{
    Operation CreateOperation();
}
```

然后加减乘除各建一个具体工厂去实现这个接口。

```
class addFactory: IFactory
{
    public Operation CreateOperation()
    {
        return new OperationAdd();
    }
}

class subFactory: IFactory
{
    public Operation CreateOperation()
    {
        return new OperationSub();
    }
}

class mulFactory: IFactory
{
    public Operation CreateOperation()
    {
        return new OperationMul();
    }
}

class divFactory: IFactory
{
    public Operation CreateOperation()
    {
        return new OperationDiv();
    }
}
```

客户端的实现是这样的。

```
IFactory operFactory = new addFactory();
operation oper = operationFactory.CreateOperation();
oper.NumberA = 1;
oper.NumberB = 2;
double result = oper.GetResult();
```

简单工厂模式的最大优点在于工厂类中包涵了必要的逻辑判断，根据客户端的选择条件动态实例化相关的类，对于客户端来说，去除了与具体相关产品的依赖。就像你的计算器，让客户端不用管该用哪个类的实例，只需要把'+'给工厂，工厂自动就给出了相应的实例，客户端只要去做运算就可以了，不同的实例会实现不同的运算。但是问题也就在这里，如你所说，如果要加一个'求M数的N次方'的功能，我们是一定需要给运算工厂类的方法里加'case'的分支条件的，修改原来的类并不是好方法，这就等于说，我们不但对扩展开放了，对修改也开放了，这样违背了开放封闭原则。

工厂方法模式，定义一个用于创建对象的接口，让子类决定实例化哪一个类。工厂方法使一个类的实例化延迟到其子类。 其实你仔细观察会发现，工厂方法模式实现时，客户端需要决定实例化哪一个工厂来实现运算类，选择判断的问题还是存在的，也就是说，工厂方法把简单工厂的内部逻辑判断移到了客户端代码来运行，你想要加功能，本来是修改工厂类的，而现在修改客户端。

原型模式(Prototype pattern)

简介

原型模式是创建型模式的一种,其特点在于通过「复制」一个已经存在的实例来返回新的实例,而不是新建实例。被复制的实例就是我们所称的「原型」，这个原型是可定制的。

原型模式多用于创建复杂的或者耗时的实例，因为这种情况下，复制一个已经存在的实例使程序运行更高效；或者创建值相等，只是命名不一样的同类数据。

Prototype原型模式是一种创建型设计模式，它主要面对的问题是：“某些结构复杂的对象”的创建工作；由于需求的变化，这些对象经常面临着剧烈的变化，但是他们却拥有比较稳定一致的接口。下面我们先来看下关联的的几种设计模式，予以区分，再来说说原型模式。

- Singleton单件模式解决的问题是：实体对象个数问题（这个现在还不太容易混）
- AbstractFactory抽象工厂模式解决的问题是：“一系列互相依赖的对象”的创建工作
- Builder生成器模式解决的问题是：“一些复杂对象”的创建工作，子对象变化较频繁，对算法相对稳定
- FactoryMethor工厂方法模式解决的问题是：某个对象的创建工作。

《设计模式》中说道：使用原型实例指定创建对象的种类，然后通过拷贝这些原型来创建新的对象。

例子

```
/** Prototype Class **/
 public class Cookie implements Cloneable {

    public Object clone() throws CloneNotSupportedException
    {
        //In an actual implementation of this pattern you would now
        //the expensive to produce parts from the copies that are h
        return (Cookie) super.clone();
    }
 }

 /** Concrete Prototypes to clone **/
 public class CoconutCookie extends Cookie { }

 /** Client Class**/
 public class CookieMachine
 {

   private Cookie cookie;//cookie必须是可复制的

     public CookieMachine(Cookie cookie) {
         this.cookie = cookie;
     }

    public Cookie makeCookie()
    {
        try
        {
            return (Cookie) cookie.clone();
        } catch (CloneNotSupportedException e)
        {
            e.printStackTrace();
        }
        return null;
    }


     public static void main(String args[]){
         Cookie tempCookie =  null;
         Cookie prot = new CoconutCookie();
         CookieMachine cm = new CookieMachine(prot); //设置原型
         for(int i=0; i<100; i++)
             tempCookie = cm.makeCookie();//通过复制原型返回多个cookie
     }
 }
```

现在我们再来看看原型模式的几个要点：

- Prototype模式同样用于隔离类对象的使用者和具体类型（易变类）之间的耦合关系，它同样要求这些“易变类”拥有“稳定的接口”。

- Prototype模式对于“如何创建易变类的实体对象”采用“原型克隆”的方法来实现，它使得我们可以非常灵活地动态创建“拥有某些稳定接口”的新对象——所需工作仅仅是注册一个新类的对象（即原型），然后在任何需要的地方不断地Clone。
- Prototype模式中的Clone方法可以利用Object类的MemberwiseClone（）或者序列化来实现深拷贝。

Javascript中的prototype就是使用了原型模式

```
function Person(name, age, job) {
  this.name = name;
  this.age = age;
  this.job = job;
}

// 通过原型模式来添加所有实例共享的方法
// sayName() 方法将会被Person的所有实例共享，而避免了重复创建
Person.prototype.sayName = function () {
  console.log(this.name);
};

var person1 = new Person('Weiwei', 27, 'Student');
var person2 = new Person('Lily', 25, 'Doctor');

console.log(person1.sayName === person2.sayName); // true

person1.sayName(); // Weiwei
person2.sayName(); // Lily
```

正如上面的代码所示，通过原型模式定义的方法sayName()为所有的实例所共享。也就是， person1和person2访问的是同一个sayName()函数。同样的，公共属性也可以使用原型模式进行定义。例如：

```
function Chinese (name) {
    this.name = name;
}

Chinese.prototype.country = 'China'; // 公共属性，所有实例共享
```

由于所有的实例对象共享同一个prototype对象，那么从外界看起来，prototype对象就好像是实例对象的原型，而实例对象则好像"继承"了prototype对象一样。它是浅拷贝。

//TODO clone的原理()

JAVA的clone()实现机制涉及到了反射、IO流操作、序列化等。

模版方法模式(Template method pattern)

简介

模板方法模式定义了一个算法的步骤，并允许子类别为一个或多个步骤提供其实践方式。让子类别在不改变算法架构的情况下，重新定义算法中的某些步骤。在软件工程中，它是一种软件设计模式，和C++模板没有关连。

实例

事实上，模版方法是编程中一个经常用到的模式。先来看一个例子，某日，程序员A拿到一个任务：给定一个整数数组，把数组中的数由小到大排序，然后把排序之后的结果打印出来。经过分析之后，这个任务大体上可分为两部分，排序和打印，打印功能好实现，排序就有点麻烦了。但是A有办法，先把打印功能完成，排序功能另找人做。

```
abstract class AbstractSort {

    /**
     * 将数组array由小到大排序
     * @param array
     */
    protected abstract void sort(int[] array);

    public void showSortResult(int[] array){
        this.sort(array);
        System.out.print("排序结果：");
        for (int i = 0; i < array.length; i++){
            System.out.printf("%3s", array[i]);
        }
    }
}
```

写完后，A找到刚毕业入职不久的同事B说：有个任务，主要逻辑我已经写好了，你把剩下的逻辑实现一下吧。于是把AbstractSort类给B，让B写实现。B拿过来一看，太简单了，10分钟搞定，代码如下：

```
class ConcreteSort extends AbstractSort {

    @Override
    protected void sort(int[] array){
        for(int i=0; i<array.length-1; i++){
            selectSort(array, i);
        }
    }

    private void selectSort(int[] array, int index) {
        int MinValue = 32767; // 最小值变量
        int indexMin = 0; // 最小值索引变量
        int Temp; // 暂存变量
        for (int i = index; i < array.length; i++) {
            if (array[i] < MinValue){ // 找到最小值
                MinValue = array[i]; // 储存最小值
                indexMin = i;
            }
        }
        Temp = array[index]; // 交换两数值
        array[index] = array[indexMin];
        array[indexMin] = Temp;
    }
}
```

写好后交给A，A拿来一运行：

```
public class Client {
    public static int[] a = { 10, 32, 1, 9, 5, 7, 12, 0, 4, 3 }; //
    public static void main(String[] args){
        AbstractSort s = new ConcreteSort();
        s.showSortResult(a);
    }
}
```

排序结果： 0 1 3 4 5 7 9 10 12 32

运行正常。行了，任务完成。没错，这就是模版方法模式。大部分刚步入职场的毕业生应该都有类似B的经历。一个复杂的任务，由公司中的牛人们将主要的逻辑写好，然后把那些看上去比较简单的方法写成抽象的，交给其他的同事去开发。这种分工方式在编程人员水平层次比较明显的公司中经常用到。比如一个项目组，有架构师，高级工程师，初级工程师，则一般由架构师使用大量的接口、抽象类将整个系统的逻辑串起来，实现的编码则根据难度的不同分别交给高级工程师和初级工程师来完成。怎么样，是不是用到过模版方法模式？

模版方法的优点及适用场景

`AbstractClass`是一个抽象类，其实也就是一个抽象模版，定义并实现一个模版的方法。这个模版的方法一般是一个具体的方法,它给出了一个顶级逻辑的骨架，而逻辑组成步骤在相应的抽象操作中，推迟到子类实现。顶级逻辑也有可能调用一些具体方法。

```
abstract class AbstractClass
{
    public abstract void PrimitiveOperation1();
    public abstract void PrimitiveOperation2();

    public void TemplateMethod()
    {
        PrimitiveOperation1();
        PrimitiveOperation2();
        console.WriteLine("");
    }
}
```

`ConcreteClass`实现夫类所定义的一个或者多个抽象方法。每一个`AbstractClass`都可以有任意多个`ConcreteClass`与之对应，而每一个`ConcreteClass`都可以给出这些抽象方法(也就是顶级逻辑的组成步骤)的不同实现，从而使得顶级逻辑的实现各不相同。

```
class ConcreteClassA: AbstractClass
{
    public override void PrimitiveOperation1()
    {
        Console.writeLine("具体类A方法1实现");
    }
        public override void PrimitiveOperation2()
    {
        Console.writeLine("具体类A方法2实现");
    }
}

class ConcreteClassB: AbstractClass
{
    public override void PrimitiveOperation1()
    {
        Console.writeLine("具体类B方法1实现");
    }
        public override void PrimitiveOperation2()
    {
        Console.writeLine("具体类B方法2实现");
    }
}
```

客户端调用

```
static void Main(string[] args)
{
    AbstractClass c;
    c = new ConcreteClassA();
    c.TemplateMethod();

    c = new ConcreteClassB();
    c.TemplateMethod();

    Console.Read();
}
```

模版方法模式是通过把不变的行为搬移到超类，去除子类中的重复代码来体现它的优势。当不变的和可变的行为在方法的子类实现中混合在一起的时候，不变的行为就会在子类中重复出现。我们通过模版方法模式把这些行为版已到单一的地方，这样就帮助子类摆脱重复的不变的行为的纠缠。

外观模式(Facade pattern)

简介

外观模式（Facade pattern），是软件工程中常用的一种软件设计模式，它为子系统中的一组接口提供一个统一的高层接口，使得子系统更容易使用。

模式实现

某软件公司欲开发一个可应用于多个软件的文件加密模块，该模块可以对文件中的数据进行加密并将加密之后的数据存储在一个新文件中，具体的流程包括三个部分，分别是读取源文件、加密、保存加密之后的文件，其中，读取文件和保存文件使用流来实现，加密操作通过求模运算实现。这三个操作相对独立，为了实现代码的独立重用，让设计更符合单一职责原则，这三个操作的业务代码封装在三个不同的类中。 现使用外观模式设计该文件加密模块。

(1) FileReader：文件读取类，充当子系统类。

```
//FileReader.cs
using System;
using System.Text;
using System.IO;

namespace FacadeSample
{
    class FileReader
    {
        public string Read(string fileNameSrc)
        {
       Console.Write("读取文件，获取明文：");
            FileStream fs = null;
            StringBuilder sb = new StringBuilder();
       try
            {
                fs = new FileStream(fileNameSrc, FileMode.Open);
                int data;
               while((data = fs.ReadByte())!= -1)
                {
            sb = sb.Append((char)data);
               }
               fs.Close();
               Console.WriteLine(sb.ToString());
       }
       catch(FileNotFoundException e)
            {
           Console.WriteLine("文件不存在！");
       }
       catch(IOException e)
            {
           Console.WriteLine("文件操作错误！");
       }
       return sb.ToString();
        }
    }
}
```

(2) CipherMachine：数据加密类，充当子系统类。

```
//CipherMachine.cs
using System;
using System.Text;

namespace FacadeSample
{
    class CipherMachine
    {
       public string Encrypt(string plainText)
       {
       Console.Write("数据加密，将明文转换为密文：");
       string es = "";
            char[] chars = plainText.ToCharArray();
       foreach(char ch in chars)
            {
                string c = (ch % 7).ToString();
           es += c;
       }
            Console.WriteLine(es);
       return es;
    }
    }
}
```

(3) FileWriter：文件保存类，充当子系统类。

```
//FileWriter.cs
using System;
using System.IO;
using System.Text;

namespace FacadeSample
{
    class FileWriter
    {
        public void Write(string encryptStr,string fileNameDes)
        {
       Console.WriteLine("保存密文，写入文件。");
            FileStream fs = null;
       try
            {
               fs = new FileStream(fileNameDes, FileMode.Create);
                byte[] str = Encoding.Default.GetBytes(encryptStr);
                fs.Write(str,0,str.Length);
                fs.Flush();
               fs.Close();
       }
       catch(FileNotFoundException e)
            {
        Console.WriteLine("文件不存在！");
       }
       catch(IOException e)
            {
                Console.WriteLine(e.Message);
           Console.WriteLine("文件操作错误！");
       }
        }
    }
}
```

(4) EncryptFacade：加密外观类，充当外观类。

```
// EncryptFacade.cs
namespace FacadeSample
{
    class EncryptFacade
    {
        //维持对其他对象的引用
         private FileReader reader;
        private CipherMachine cipher;
        private FileWriter writer;

        public EncryptFacade()
        {
            reader = new FileReader();
            cipher = new CipherMachine();
            writer = new FileWriter();
        }

        //调用其他对象的业务方法
         public void FileEncrypt(string fileNameSrc, string fileNar
        {
            string plainStr = reader.Read(fileNameSrc);
            string encryptStr = cipher.Encrypt(plainStr);
            writer.Write(encryptStr, fileNameDes);
        }
    }
}
```

(5) Program：客户端测试类

```
//Program.cs
using System;

namespace FacadeSample
{
    class Program
    {
        static void Main(string[] args)
        {
            EncryptFacade ef = new EncryptFacade();
            ef.FileEncrypt("src.txt", "des.txt");
            Console.Read();
        }
    }
}
```

结果及分析

```
读取文件，获取明文：Hello world!
数据加密，将明文转换为密文：233364062325
保存密文，写入文件。
```

抽象外观类

在标准的外观模式结构图中，如果需要增加、删除或更换与外观类交互的子系统类，必须修改外观类或客户端的源代码，这将违背开闭原则，因此可以通过引入抽象外观类来对系统进行改进，在一定程度上可以解决该问题。在引入抽象外观类之后，客户端可以针对抽象外观类进行编程，对于新的业务需求，不需要修改原有外观类，而对应增加一个新的具体外观类，由新的具体外观类来关联新的子系统对象，同时通过修改配置文件来达到不修改任何源代码并更换外观类的目的。

下面通过一个具体实例来学习如何使用抽象外观类：

如果在应用实例“文件加密模块”中需要更换一个加密类，不再使用原有的基于求模运算的加密类CipherMachine，而改为基于移位运算的新加密类NewCipherMachine，其代码如下：

```
using System;

namespace FacadeSample
{
    class NewCipherMachine
    {
        public string Encrypt(string plainText)
        {
            Console.Write("数据加密，将明文转换为密文：");
            string es = "";
            int key = 10;//设置密钥，移位数为10
            char[] chars = plainText.ToCharArray();
            foreach(char ch in chars)
            {
                int temp = Convert.ToInt32(ch);
                //小写字母移位
                if (ch >= 'a' && ch <= 'z') {
                    temp += key % 26;
                    if (temp > 122) temp -= 26;
                    if (temp < 97) temp += 26;
                }
                //大写字母移位
                if (ch >= 'A' && ch <= 'Z') {
                    temp += key % 26;
                    if (temp > 90) temp -= 26;
                    if (temp < 65) temp += 26;
                }
                es += ((char)temp).ToString();
            }
            Console.WriteLine(es);
            return es;
        }
    }
}
```

如果不增加新的外观类，只能通过修改原有外观类EncryptFacade的源代码来实现加密类的更换，将原有的对CipherMachine类型对象的引用改为对NewCipherMachine类型对象的引用，这违背了开闭原则，因此需要通过增加新的外观类来实现对子系统对象引用的改变。

如果增加一个新的外观类NewEncryptFacade来与FileReader类、FileWriter类以及新增加的NewCipherMachine类进行交互，虽然原有系统类库无须做任何修改，但是因为客户端代码中原来针对EncryptFacade类进行编程，现在需要改为NewEncryptFacade类，因此需要修改客户端源代码。

如何在不修改客户端代码的前提下使用新的外观类呢？解决方法之一是：引入一个抽象外观类，客户端针对抽象外观类编程，而在运行时再确定具体外观类，引入抽象外观类之后的文件加密模块结构图如图5所示：

客户类Client针对抽象外观类AbstractEncryptFacade进行编程，AbstractEncryptFacade代码如下：

```
namespace FacadeSample
{
    abstract class AbstractEncryptFacade
    {
        public abstract void FileEncrypt(string fileNameSrc, strin
    }
}
```

新增具体加密外观类NewEncryptFacade代码如下：

```
namespace FacadeSample
{
    class NewEncryptFacade : AbstractEncryptFacade
    {
        private FileReader reader;
        private NewCipherMachine cipher;
        private FileWriter writer;

        public NewEncryptFacade()
        {
            reader = new FileReader();
            cipher = new NewCipherMachine();
            writer = new FileWriter();
        }

        public override void FileEncrypt(string fileNameSrc, strin
        {
            string plainStr = reader.Read(fileNameSrc);
            string encryptStr = cipher.Encrypt(plainStr);
            writer.Write(encryptStr, fileNameDes);
        }
    }
}
```

```
配置文件App.config中存储了具体外观类的类名，代码如下：
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <appSettings>
    <add key="facade" value="FacadeSample.NewEncryptFacade"/>
  </appSettings>
</configuration>
```

客户端测试代码修改如下：

```
using System;
using System.Configuration;
using System.Reflection;

namespace FacadeSample
{
    class Program
    {
        static void Main(string[] args)
        {
            AbstractEncryptFacade ef; //针对抽象外观类编程
            //读取配置文件
            string facadeString = ConfigurationManager.AppSettings[
            //反射生成对象
            ef = (AbstractEncryptFacade)Assembly.Load("FacadeSample
            ef.FileEncrypt("src.txt", "des.txt");
            Console.Read();
        }
    }
}
```

模式适用场景

在以下情况下可以考虑使用外观模式：

1. 当要为访问一系列复杂的子系统提供一个简单入口时可以使用外观模式。
2. 客户端程序与多个子系统之间存在很大的依赖性。引入外观类可以将子系统与客户端解耦，从而提高子系统的独立性和可移植性。
3. 在层次化结构中，可以使用外观模式定义系统中每一层的入口，层与层之间不直接产生联系，而通过外观类建立联系，降低层之间的耦合度。

建造者模式(Builder Pattern)

简介

生成器模式是一种对象构建模式。它可以将复杂对象的建造过程抽象出来（抽象类别），使这个抽象过程的不同实现方法可以构造出不同表现（属性）的对象。

范例

```
/** "Product" */
class Pizza {
  private String dough = "";
  private String sauce = "";
  private String topping = "";

  public void setDough (String dough)     { this.dough = dough; }
  public void setSauce (String sauce)     { this.sauce = sauce; }
  public void setTopping (String topping) { this.topping = topping
}


''/** "Abstract Builder" */''
abstract class PizzaBuilder {
  protected Pizza pizza;

  public Pizza getPizza() { return pizza; }
  public void createNewPizzaProduct() { pizza = new Pizza(); }

  public abstract void buildDough();
  public abstract void buildSauce();
  public abstract void buildTopping();
}

/** "ConcreteBuilder" */
class HawaiianPizzaBuilder extends PizzaBuilder {
  public void buildDough()   { pizza.setDough("cross"); }
  public void buildSauce()   { pizza.setSauce("mild"); }
  public void buildTopping() { pizza.setTopping("ham+pineapple");
}

/** "ConcreteBuilder" */
class SpicyPizzaBuilder extends PizzaBuilder {
  public void buildDough()   { pizza.setDough("pan baked"); }
  public void buildSauce()   { pizza.setSauce("hot"); }
  public void buildTopping() { pizza.setTopping("pepperoni+salami'
}
```

```
''/** "Director" */''
class Waiter {
  private PizzaBuilder pizzaBuilder;

  public void setPizzaBuilder (PizzaBuilder pb) { pizzaBuilder = p
  public Pizza getPizza() { return pizzaBuilder.getPizza(); }

  public void constructPizza() {
    pizzaBuilder.createNewPizzaProduct();
    pizzaBuilder.buildDough();
    pizzaBuilder.buildSauce();
    pizzaBuilder.buildTopping();
  }
}

/** A customer ordering a pizza. */
class BuilderExample {
  public static void main(String[] args) {
    Waiter waiter = new Waiter();
    PizzaBuilder hawaiian_pizzabuilder = new HawaiianPizzaBuilder(
    PizzaBuilder spicy_pizzabuilder = new SpicyPizzaBuilder();

    waiter.setPizzaBuilder ( hawaiian_pizzabuilder );
    waiter.constructPizza();

    Pizza pizza = waiter.getPizza();
  }
}
```

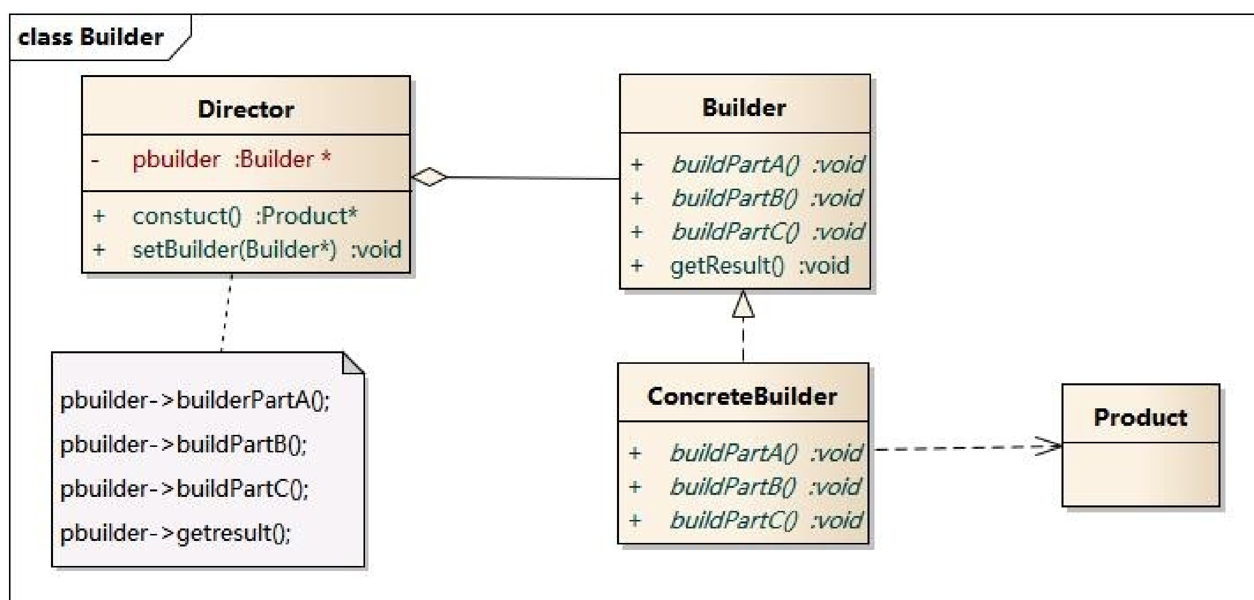
抽象工厂模式与生成器相似，因为它也可以创建复杂对象。主要的区别是生成器模式着重于一步步构造一个复杂对象。而抽象工厂模式着重于多个系列的产品对象（简单的或是复杂的）。生成器在最后的一步返回产品，而对于抽象工厂来说，产品是立即返回的。

实例

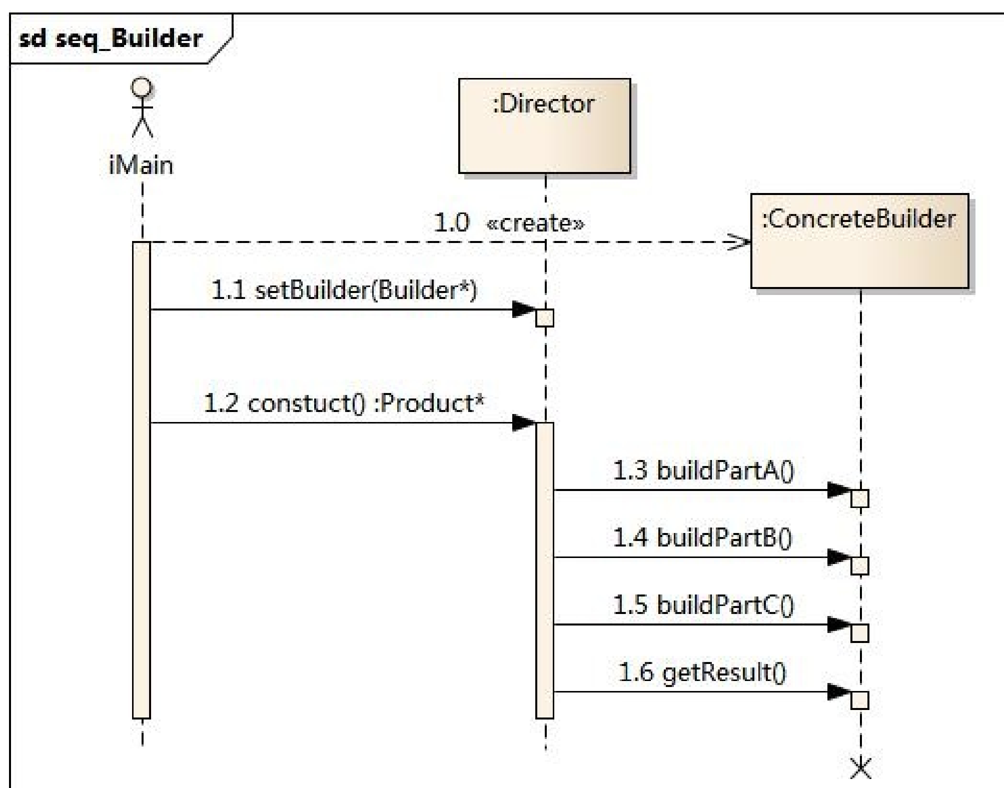
。

建造者模式包含如下角色：

- Builder：抽象建造者
- ConcreteBuilder：具体建造者
- Director：指挥者
- Product：产品角色



实序图



建造者模式主要是用于创建一些复杂的对象，这些对象内部构建间的建造顺序是稳定的，但是对象内部的构建通常面临着复杂的变化。

建造者模式的好处就是使得建造代码与表示代码分离，由于建造者隐藏了改产品是如何组装的，所以若需要改变一个产品的内部表示，只需要再定义一个具体的建造者就可以了。

观察者模式(Observer Pattern)

简介

观察者模式（有时又被称为发布/订阅模式）是软件设计模式的一种。在此种模式中，一个目标对象管理所有相依于它的观察者对象，并且在它本身的状态改变时主动发出通知。这通常透过呼叫各观察者所提供的方法来实现。此种模式通常被用来实时事件处理系统。



抽象目标类别

此抽象类别提供一个界面让观察者进行添附与解附作业。此类别内有个不公开的观察者串炼，并透过下列函式(方法)进行作业

- 添附(Attach)：新增观察者到串炼内，以追踪目标对象的变化。
- 解附(Detach)：将已经存在的观察者从串炼中移除。
- 通知(Notify)：利用观察者所提供的更新函式来通知此目标已经产生变化。

添附函式包涵了一个观察者对象参数。也许是观察者类别的虚拟函式(即更新函式)，或是在非面向对象的设定中所使用的函式指标(更广泛来讲，函式子或是函式对象)。

目标类别

此类别提供了观察者欲追踪的状态。也利用其源类别(例如前述的抽象目标类别)所提供的方法,来通知所有的观察者其状态已经更新。此类别拥有以下函式

- 取得状态(GetState)：回传该目标对象的状态。

抽象观察者界面

抽象观察者类别是一个必须被实做的抽象类别。这个类别定义了所有观察者都拥有的更新用界面，此界面是用来接收目标类别所发出的更新通知。此类别含有以下函式

- 更新(Update)：会被实做的一个抽象(虚拟)函式。

观察者类别

这个类别含有指向目标类别的参考(reference)，以接收来自目标类别的更新状态。此类别含有以下函式

- 更新(Update)：是前述抽象函式的实做。当这个函式被目标对象呼叫时，观察

者对象将会呼叫目标对象的取得状态函式，来其所拥有的更新目标对象资讯。每个观察者类别都要实做它自己的更新函式，以应对状态更新的情形。

当目标对象改变时，会通过呼叫它自己的通知函式来将通知送给每一个观察者对象，这个通知函式则会去呼叫已经添附在串炼内的观察者更新函式。通知与更新函式可能会有一些参数，好指明是目前目标对象内的何种改变。这么作将可增进观察者的效率(只更新那些改变部分的状态)。

用途

- 当抽象个体有两个互相依赖的层面时。封装这些层面在单独的对象内将可允许程序员单独地去变更与重复使用这些对象，而不会产生两者之间交互的问题。
- 当其中一个对象的变更会影响其他对象，却又不知道多少对象必须被同时变更时。
- 当对象应该有能力通知其他对象，又不应该知道其他对象的实做细节时。

```
=== C++ ===
<source lang="cpp">
#include <list>
#include <vector>
#include <algorithm>
#include <iostream>
using namespace std;

// The Abstract Observer
class ObserverBoardInterface
{
public:
    virtual void update(float a,float b,float c) = 0;
};

// Abstract Interface for Displays
class DisplayBoardInterface
{
public:
    virtual void show() = 0;
};

// The Abstract Subject
class WeatherDataInterface
{
public:
    virtual void registerob(ObserverBoardInterface* ob) = 0;
    virtual void removeob(ObserverBoardInterface* ob) = 0;
    virtual void notifyOb() = 0;
};

// The Concrete Subject
class ParaWeatherData: public WeatherDataInterface
{
public:
```

```
    void SensorDataChange(float a,float b,float c)
    {
        m_humidity = a;
        m_temperature = b;
        m_pressure = c;
        notifyOb();
    }

    void registerob(ObserverBoardInterface* ob)
    {
        m_obs.push_back(ob);
    }

    void removeob(ObserverBoardInterface* ob)
    {
        m_obs.remove(ob);
    }
protected:
    void notifyOb()
    {
        list<ObserverBoardInterface*>::iterator pos = m_obs.begin()
        while (pos != m_obs.end())
        {
            ((ObserverBoardInterface* )(*pos))->update(m_humidity,r
            (dynamic_cast<DisplayBoardInterface*>(*pos))->show();
            ++pos;
        }
    }

private:
    float        m_humidity;
    float        m_temperature;
    float        m_pressure;
    list<ObserverBoardInterface* > m_obs;
};

// A Concrete Observer
class CurrentConditionBoard : public ObserverBoardInterface, public
{
public:
    CurrentConditionBoard(WeatherDataInterface& a):m_data(a)
    {
        m_data.registerob(this);
    }
    void show()
    {
        cout<<"_____CurrentConditionBoard_____"<<endl;
        cout<<"humidity: "<<m_h<<endl;
        cout<<"temperature: "<<m_t<<endl;
        cout<<"pressure: "<<m_p<<endl;
        cout<<"_______________________________"<<endl;
    }
```

```
    void update(float h, float t, float p)
    {
        m_h = h;
        m_t = t;
        m_p = p;
    }

private:
    float m_h;
    float m_t;
    float m_p;
    WeatherDataInterface& m_data;
};

// A Concrete Observer
class StatisticBoard : public ObserverBoardInterface, public Displa
{
public:
    StatisticBoard(WeatherDataInterface& a):m_maxt(-1000),m_mint(10
    {
        m_data.registerob(this);
    }

    void show()
    {
        cout<<"________StatisticBoard_________"<<endl;
        cout<<"lowest  temperature: "<<m_mint<<endl;
        cout<<"highest temperature: "<<m_maxt<<endl;
        cout<<"average temperature: "<<m_avet<<endl;
        cout<<"_______________________________"<<endl;
    }

    void update(float h, float t, float p)
    {
        ++m_count;
        if (t>m_maxt)
        {
            m_maxt = t;
        }
        if (t<m_mint)
        {
            m_mint = t;
        }
        m_avet = (m_avet * (m_count-1) + t)/m_count;
    }

private:
    float m_maxt;
    float  m_mint;
    float m_avet;
    int m_count;
    WeatherDataInterface& m_data;
};
```

```
int main(int argc, char *argv[])
{

    ParaWeatherData * wdata = new ParaWeatherData;
    CurrentConditionBoard* currentB = new CurrentConditionBoard(*wd
    StatisticBoard* statisticB = new StatisticBoard(*wdata);

    wdata->SensorDataChange(10.2, 28.2, 1001);
    wdata->SensorDataChange(12, 30.12, 1003);
    wdata->SensorDataChange(10.2, 26, 806);
    wdata->SensorDataChange(10.3, 35.9, 900);

    wdata->removeob(currentB);

    wdata->SensorDataChange(100, 40, 1900);

    delete statisticB;
    delete currentB;
    delete wdata;

    return 0;
}
```

将一个系统分割成一系列相互作用的类有个很不好的副作用，那就是需要维护相关对象间的一致性。我们不希望为了维持一致性而逝各类紧密耦合，这样会给维护，扩展和重用都带来不便。

什么时候考虑使用观察者模式呢。

当一个对象的改变需要同事改变其它对象而且它不知道具体有多少对象有待改变时应该考虑使用观察者模式。

一个抽象模型有两个方面，其中一方面依赖于另一方面，这时用观察者模式可以将这两者防撞在独立的对象中使它们各自独立地改变和服用。

总的来讲，观察者模式所做的工作其实就是在解除耦合，让耦合的双方都依赖于抽象，而不是依赖于具体。从而使得各自的变化都不会影响另一个的变化。

抽象工厂模式(Abstract factory Pattern)

简介

抽象工厂模式是一种软件开发设计模式。抽象工厂模式提供了一种方式，可以将一组具有同一主题的单独的工厂封装起来。在正常使用中，客户端程序需要创建抽象工厂的具体实现，然后使用抽象工厂作为接口来创建这一主题的具体对象。客户端程序不需要知道（或关心）它从这些内部的工厂方法中获得对象的具体类型，因为客户端程序仅使用这些对象的通用接口。抽象工厂模式将一组对象的实现细节与他们的一般使用分离开来。

简例

有个项目原来是依赖于access数据库，现在要用sqlserver，怎么更改呢(解耦)？

用工厂方法模式的数据访问程序

IUser接口，用于客户端访问，解除与具体数据库访问的耦合。

```
interface IUser
{
    void Insert(User user);

    User GetUser(int id);
}
```

Sqlserver类，用于访问SQL Server的User。

```
class SqlserverUser : IUser
{
    public void Insert(User user)
    {
        Console.WriteLine("在SQL Server中给User表增加一条记录");
    }

    public User GetUser(int id)
    {
        Console.WriteLine("在SQL Server中给User表获取一条记录");
        return null;
    }

}
```

AccessUser类，用于访问Access的User。

```
class AccessUser : IUser
{
    public void Insert(User user)
    {
        Console.WriteLine("在access中给User表增加一条记录");
    }

    public User GetUser(int id)
    {
        Console.WriteLine("在access中给User表获取一条记录");
        return null;
    }

}
```

IFactory接口，定义一个创建访问User表对象的抽象的工厂接口。

```
interface Ifactory
{
    IUser CreateUser();
}
```

sqlServerFactory类，实现IFactory接口，实例化SqlserverUser。

```
class sqlServerFactory:  IFactory
{
    public IUser CreateUser()
    {
        return new SqlserverUser();
    }
}
```

AccessFactory类，实现IFactory接口，实例化AccessUser.

```
class AccessFactory:  IFactory
{
    public IUser CreateUser()
    {
        return new AccessUser();
    }
}
```

客户端代码

```
static void main(string[] args)
{
    User user = new User();

    IFactory factory = new SqlServerFactory();

    IUser iu = factory.CreateUser();

    iu.Insert(user);
    iu.GetUser(1);
    Console.Read();
}
```

但是数据里面不可能只有一个User表，比如说增加部门表，此时怎么办呢。

抽象工厂类

IUser接口，用于客户端访问，解除与具体数据库访问的耦合。

```
interface IDepartment
{
    void Insert(Department department);
    Department GetDepartment(int id);
}
```

SqlserverDepartment类，用于访问SQL Server的Department。

```
class SqlserverDepartment : IDepartment
{
    public void Insert(Department department)
    {
        Console.WriteLine("在SQL Server中给Department表增加一条记录");
    }

    public Department GetDepartment(int id)
    {
        Console.WriteLine("在SQL Server中给Department表获取一条记录");
        return null;
    }

}
```

AccessDepartment类，用于访问Access的Department。

```
class AccessDepartment : IDepartment
{
    public void Insert(Department department)
    {
        Console.WriteLine("在Access中给Department表增加一条记录");
    }

    public Department GetDepartment(int id)
    {
        Console.WriteLine("在Access中给Department表获取一条记录");
        return null;
    }

}
```

```
interface IUser
{
    void Insert(User user);

    User GetUser(int id);
}
```

Sqlserver类，用于访问SQL Server的User。

```
class SqlserverUser : IUser
{
    public void Insert(User user)
    {
        Console.WriteLine("在SQL Server中给User表增加一条记录");
    }

    public User GetUser(int id)
    {
        Console.WriteLine("在SQL Server中给User表获取一条记录");
        return null;
    }

}
```

AccessUser类，用于访问Access的User。

```
class AccessUser : IUser
{
    public void Insert(User user)
    {
        Console.WriteLine("在access中给User表增加一条记录");
    }

    public User GetUser(int id)
    {
        Console.WriteLine("在access中给User表获取一条记录");
        return null;
    }

}
```

IFactory接口，定义一个创建访问User表对象的抽象的工厂接口。

```
interface Ifactory
{
    IUser CreateUser();

    IDepartment CreateDepartment();
}
```

sqlServerFactory类，实现IFactory接口，实例化SqlserverUser。

```
class sqlServerFactory:  IFactory
{
    public IUser CreateUser()
    {
        return new SqlserverUser();
    }

    public IDepartment CreateDepartment()
    {
        return new SqlserverDepartment();
    }

}
```

AccessFactory类，实现IFactory接口，实例化AccessUser.

```
class AccessFactory:  IFactory
{
    public IUser CreateUser()
    {
        return new AccessUser();
    }

    public IDepartment CreateDepartment()
    {
        return new AccessDepartment();
    }
}
```

客户端代码

```
static void main(string[] args)
{
    User user = new User();

    IFactory factory = new SqlServerFactory();

    IUser iu = factory.CreateUser();

    iu.Insert(user);
    iu.GetUser(1);
    Console.Read();
}
```

抽象工厂模式：提供一个创建一系列相关或者相互依赖的对象的接口，而无需指定它们具体的类。

用简单工厂来改进抽象工厂

```
class DataAccess
{
    private static readonly string db = "Sqlserver";
    //    private static readonly string db = "access";

    public static IUser CreateUser()
    {
        IUser result = null;

        switch (db)
        {
            case:"Sqlserver":
                result = new SqlserverUser();
                break;
            case:"Acesss":
                result = new AcesssUser();
                break;
        }
        return result;
    }

    publci static IDepartment CreateDepartment()
    {
        IDepartment result = null;
                switch (db)
        {
            case:"Sqlserver":
                result = new SqlserverDepartment();
                break;
            case:"Acesss":
                result = new AcesssDepartment();
                break;
        }
        return result;
    }

}
```

客户端代码

```
static void Main(string[] args)
{
    User user = new User();
    Department dept = new Department();

    IUser iu = DataAccess.createUser();
    iu.Insert(user);
    iu.GetUser(1);

    IDepartment id = DataAccess.createDepartment();
    id.Inert(dept);
    id.GetDepartment(1);

    Console.Read();
}
```

利用反射或者配置文件都可以减少在所有简单工厂的switch和if，解除分支判断带来的耦合。

“工厂”是创建产品（对象）的地方，其目的是将产品的创建与产品的使用分离。抽象工厂模式的目的，是将若干抽象产品的接口与不同主题产品的具体实现分离开。这样就能在增加新的具体工厂的时候，不用修改引用抽象工厂的客户端代码。

使用抽象工厂模式，能够在具体工厂变化的时候，不用修改使用工厂的客户端代码，甚至是在运行时。然而，使用这种模式或者相似的设计模式，可能给编写代码带来不必要的复杂性和额外的工作。正确使用设计模式能够抵消这样的“额外工作”。

从NSArray看类簇

Class Clusters（类簇）是抽象工厂模式在iOS下的一种实现，众多常用类，如NSString，NSArray，NSDictionary，NSNumber都运作在这一模式下，它是接口简单性和扩展性的权衡体现，在我们完全不知情的情况下，偷偷隐藏了很多具体的实现类，只暴露出简单的接口。

虽然官方文档中拿NSNumber说事儿，但Foundation并没有像图中描述的那样为每个number都弄一个子类，于是研究下NSArray类簇的实现方式。

__NSPlacehodlerArray

熟悉这个模式的同学很可能看过下面的测试代码，将原有的alloc+init拆开写：

```
id obj1 = [NSArray alloc]; // __NSPlacehodlerArray *
id obj2 = [NSMutableArray alloc];  // __NSPlacehodlerArray *
id obj3 = [obj1 init];  // __NSArrayI *
id obj4 = [obj2 init];  // __NSArrayM *
```

发现+ alloc后并非生成了我们期望的类实例，而是一个**NSPlacehodlerArray**的中间对象，后面的**- init**或**- initWithXXXXX**消息都是发送给这个中间对象，再由它做工厂，生成真的对象。这里的NSArrayI和__NSArrayM分别对应Immutable和Mutable（后面的I和M的意思）

于是顺着思路猜实现，__NSPlacehodlerArray必定用某种方式存储了它是由谁alloc出来的这个信息，才能在init的时候知道要创建的是可变数组还是不可变数组

经过研究发现，Foundation用了一个很贱的比较静态实例地址方式来实现，伪代码如下：

```
static __NSPlacehodlerArray *GetPlaceholderForNSArray() {
    static __NSPlacehodlerArray *instanceForNSArray;
    if (!instanceForNSArray) {
        instanceForNSArray = [[__NSPlacehodlerArray alloc] init];
    }
    return instanceForNSArray;
}

static __NSPlacehodlerArray *GetPlaceholderForNSMutableArray() {
    static __NSPlacehodlerArray *instanceForNSMutableArray;
    if (!instanceForNSMutableArray) {
        instanceForNSMutableArray = [[__NSPlacehodlerArray alloc] i
    }
    return instanceForNSMutableArray;
}
// NSArray实现
+ (id)alloc {
    if (self == [NSArray class]) {
        return GetPlaceholderForNSArray()
    }
}
// NSMutableArray实现
+ (id)alloc {
    if (self == [NSMutableArray class]) {
        return GetPlaceholderForNSMutableArray()
    }
}
// __NSPlacehodlerArray实现
- (id)init {
    if (self == GetPlaceholderForNSArray()) {
        self = [[__NSArrayI alloc] init];
    }
    else if (self == GetPlaceholderForNSMutableArray()) {
        self = [[__NSArrayM alloc] init];
    }
    return self;
}
```

Foundation不是开源的，所以上面的代码是猜测的，思路大概就是这样，可以这样验证下：

```
id obj1 = [NSArray alloc];
id obj2 = [NSArray alloc];
id obj3 = [NSMutableArray alloc];
id obj4 = [NSMutableArray alloc];
// 1和2地址相同，3和4地址相同，无论多少次都相同，且地址相差16位
```

状态模式(State Pattern)

简介

状态模式，当一个对象在内在状态改变时，允许改变起行为，这个对象看起来像是改变了其类。

状态模式主要解决的是当控制一个对象状态转换的条件表达式过于复杂的情况。把状态的判断逻辑转移到表示不同状态的一系列类中，可以把复杂的判断逻辑简化。

简例

工作状态－分类板

```
public class Work
{
    private int hour;
    public int Hour
    {
        get {return hour;}
        set {hour = value;}
    }

    private bool finish = false;
    public bool TaskFinished
    {
        get {return finish;}
        set { finish = value;}
    }

    public void WriteProgram()
    {

        if (hour < 12)
        {
            Console.WriteLine("当前时间:{0}点，上午"，hour);
        } else if (hour < 13)
        {
            Console.WriteLine("当前时间:{0}点，午饭"，hour);
        } else if (hour < 17)
        {
            Console.WriteLine("当前时间:{0}点，下午"，hour);
        } else {
            if (finish) {
            Console.WriteLine("当前时间:{0}点，下班回家"，hour);
            } else {
                if (hour < 21)
                {
                    Console.WriteLine("当前时间:{0}点，加班哦，疲累之极
                } else {
                    Console.WriteLine("当前时间:{0}点，不行了，睡着了"
                }
            }
        }
    }
}
```

客户端程序如下

```
static void Main(string[] args)
{
    //紧急项目
    Work emergencyProjects = new Work();
    emergencyProjects.Hour = 9;
    emergencyProjects.WriteProgram();
    emergencyProjects.Hour = 10;
    emergencyProjects.WriteProgram();
    emergencyProjects.Hour = 12;
    emergencyProjects.WriteProgram();
    emergencyProjects.Hour = 13;
    emergencyProjects.WriteProgram();
    emergencyProjects.Hour = 14;
    emergencyProjects.WriteProgram();
    emergencyProjects.Hour = 17;
    emergencyProjects.WriteProgram();

    //emergencyProjects.WorkFinished = true;
    emergencyProjects.TaskFinished = false;

    emergencyProjects.WriteProgram();
    emergencyProjects.Hour = 19;
    emergencyProjects.WriteProgram();
    emergencyProjects.Hour = 22;
    emergencyProjects.WriteProgram();

    Console.Read();

}
```

结果

此段代码有很大的问题。MartinFowler曾在《重构》中写过一个很重要的代码味道，叫做'Long Method',方法如果过长其实极有可能有坏味道了。

'Work'类的'WriteProgram'方法很长，而且有很多判断的分支，这也就意味着它的责任过大了。无论是任何状态，都需要通过它来改变，这实际上是很糟糕的。

面对对象设计其实就是希望做到代码的责任分解。这个类违背了'单一职责原则'。而且由于'WriteProgram'的方法里面有这么多判断，使得任何需求的改动或增加，都需要更改这个方法。又违背了'开放—封闭原则'。这类有个解决方案，就是'状态模式'。

状态模式的好处是将与特定状态相关的行为局部化，并且将不同状态的行为分割开来。

将特定的状态相关的行为都放入一个对象中，由于所有与状态相关的代码都存在于摸个***ConcreteState***中，所以通过定义新的字类可以很容易地增加新的状态与转换。这样做的目的就是为了消除庞大的条件分支语句。状态模式通过把各种状态转移逻辑分布到***State***的子类之间，来减少相互间的依赖。

当一个对象的行为取决于它的状态，并且它必须在运行时刻根据状态改变它的行为时，就可以考虑使用状态模式了。

工作状态—状态模式板

```
//抽象状态

public abstract class State
{
    public abstract void WriteProgram(Work w);
}
```

上午工作状态

```
public class ForenoonState: State
{
    public override void WriteProgram (Work w)
    {
        if (w.hour < 12)
        {
            Console.WriteLine("当前时间:{0}点 上午"，w.Hour);
        } else
        {
            w.SetState(new NoonState());
            w.WriteProgram();
        }
    }
}
```

中午工作状态

```
public class NoonState: State
{
    public override void WriteProgram (Work w)
    {
        if (w.hour < 13)
        {
            Console.WriteLine("当前时间:{0}点 中午"，w.Hour);
        } else
        {
            w.SetState(new AfterNoonState());
            w.WriteProgram();
        }
    }
}
```

下午工作状态

```
public class AfterNoonState: State
{
    public override void WriteProgram (Work w)
    {
        if (w.hour < 13)
        {
            Console.WriteLine("当前时间:{0}点 下午"，w.Hour);
        } else
        {
            w.SetState(new EveningState());
            w.WriteProgram();
        }
    }
}
```

晚间工作状态

```
public class EveningState: State
{
    public override void WriteProgram (Work w)
    {
        if (w.TaskFinished)
        {
            w.SetState(new RestState());
            w.WriteProgeam();
        } else
        {
            if (w.Hour < 21)
            {
                Console.WriteLine("当前时间:{0}点 加班哦，疲累"，w.Hou
            } else
            {
                w.SetState(new SleepState());
                w.WriteProgram();
            }

        }

    }
}
```

睡眠状态

```
public class SleepingState: State
{
    public override void WriteProgram (Work w)
    {
        if (w.hour < 13)
        {
                Console.WriteLine("当前时间:{0}点 不行了，睡着了。"，w.
        }
    }
}
```

下班休息状态

```
public class RestState: State
{
    public override void WriteProgram(Work w)
    {
        Console.WriteLine("当前时间:{0}点 下班回家了。"，w.Hour);
    }
}
```

工作类，此时没有了过长的分支判断语句

```
public class Work
{
    private State current;
    public Work()
    {
        current = new ForenoonState();
    }

    private double hour;
    public double Hour
    {
        get {return hour;}
        set {hour = value;}
    }

    private bool finish = false;
    public bool TaskFinshed
    {
        get { return finsh;}
        set { finished = value;}
    }

    public void SetState(State s)
    {
        current = s;
    }

    public void WriteProgram()
    {
        current.WriteProgram(this);
    }

}
```

客户端代码，没有任何改动。但我们的程序却更加灵活易变了。

```
static void Main(string[] args)
{
    //紧急项目
    Work emergencyProjects = new Work();
    emergencyProjects.Hour = 9;
    emergencyProjects.WriteProgram();
    emergencyProjects.Hour = 10;
    emergencyProjects.WriteProgram();
    emergencyProjects.Hour = 12;
    emergencyProjects.WriteProgram();
    emergencyProjects.Hour = 13;
    emergencyProjects.WriteProgram();
    emergencyProjects.Hour = 14;
    emergencyProjects.WriteProgram();
    emergencyProjects.Hour = 17;
    emergencyProjects.WriteProgram();

    //emergencyProjects.WorkFinished = true;
    emergencyProjects.TaskFinished = false;

    emergencyProjects.WriteProgram();
    emergencyProjects.Hour = 19;
    emergencyProjects.WriteProgram();
    emergencyProjects.Hour = 22;
    emergencyProjects.WriteProgram();

    Console.Read();

}
```

总结与分析

状态模式的主要优点在于封装了转换规则，并枚举可能的状态，它将所有与某个状态有关的行为放到一个类中，并且可以方便地增加新的状态，只需要改变对象状态即可改变对象的行为，还可以让多个环境对象共享一个状态对象，从而减少系统中对象的个数；其缺点在于使用状态模式会增加系统类和对象的个数，且状态模式的结构与实现都较为复杂，如果使用不当将导致程序结构和代码的混乱，对于可以切换状态的状态模式不满足“开闭原则”的要求。

适配器模式(Adapter Pattern)

简介

在设计模式中，适配器模式有时候也称包装样式或者包装(wrapper)。将一个类的接口转接成用户所期待的。一个适配使得因接口不兼容而不能在一起工作的类工作在一起，做法是将类自己的接口包裹在一个已存在的类中。

适配器模式包含两种，一种是类适配器，另一种是对象适配器。`类适配器`是通过类的继承实现的适配，而`对象适配器`是通过对象间的关联关系，组合关系实现的适配。二者在实际项目中都会经常用到，由于对象适配器是通过类间的关联关系进行耦合的，因此在设计时就可以做到比较灵活，而类适配器就只能通过覆写源角色的方法进行拓展，在实际项目中，对象适配器使用到的场景相对较多。在iOS开发中也推荐多使用组合关系，而尽量减少继承关系，这是一种很好的编程习惯，因此我在这里只介绍对象适配器，想了解更多的关于类适配器的话，请自行Google之。

```
Convert the interface of a class into another interface clients exp
将一个类的接口变成客户端所期待的另一种接口，从而使原本因接口不匹配而无法在一起工
```

简例

适配器模式说明

- Target目标角色 该角色定义把其他类转换为何种接口，也就是我们的期望接口。
- Adaptee源角色 你想把“谁”转换成目标角色，这个“谁”就是源角色，它是已经存在的、运行良好的类或对象。
- Adapter适配器角色 适配器模式的核心角色，其他两个角色都是已经存在的角色，而适配器角色是需要新建立的，他的职责非常简单：把源角色转换为目标角色。

适配器模式优点

- 适配器模式可以让两个没有任何关系的类在一起运行，只要适配器这个角色能够搞定他们就成。
- 增加了类的透明性。我们访问的是目标角色，但是实现却在源角色里。
- 提高了类的复用度。源角色在原有系统中还是可以正常使用的。
- 灵活性非常好。不想要适配器时，删掉这个适配器就好了，其他代码不用改。

Target

```
#import <Foundation/Foundation.h>

@protocol Target <NSObject>

- (void)userExpectInterface;

@end
```

Adaptee

```
#import <Foundation/Foundation.h>

@interface Adaptee : NSObject

- (void)doSometing;

@end

@implementation Adaptee

- (void)doSometing
{
    NSLog(@"adaptee doing something!");
}

@end
```

Adapter

```
#import "Target.h"
#import "Adaptee.h"

@interface Adapter : NSObject<Target>

@property (strong, nonatomic) Adaptee *adaptee;

- (id)initWithAdaptee:(Adaptee *)adaptee;

@end

@implementation Adapter

@synthesize adaptee = _adaptee;


- (id)initWithAdaptee:(Adaptee *)adaptee
{
    if (self = [super init]) {
        _adaptee = adaptee;
    }
    return self;
}

- (void)userExpectInterface
{
    [self.adaptee doSometing];
}

@end
```

main

```
#import <Foundation/Foundation.h>
#import "Adapter.h"
#import "Adaptee.h"
#import "Target.h"

int main(int argc, const char * argv[])
{

    @autoreleasepool {

        Adaptee *adaptee = [[Adaptee alloc]init];
        id<Target> object = [[Adapter alloc]initWithAdaptee:adaptee

        [object userExpectInterface];

    }
    return 0;
}
```

何时用适配器模式？两个类所做的事情相同或者相似，但是具有不同的接口时要使用它。 客户端代码可以统一调用接口就行了，这样应该可以更简单，更直接，更紧凑。在双方都不太容易修改的时候再食用适配器模式适配。

备忘录模式(Memento Pattern)

简介

备忘录模式有两个目标：

- 储存系统关键对象的重要状态；
- 维护关键对象的封装。

单一职责原则告诉我们，设计时不要把保持状态的工作和关键对象混在一起。这个专门掌握状态的对象，就称为备忘录。

备忘录模式提供了一种状态恢复的实现机制，使得用户可以方便地回到一个特定的历史步骤，当新的状态无效或者存在问题时，可以使用存储起来的备忘录将状态复原，当前很多软件都提供了Undo（撤销）操作功能，就使用了备忘录模式。



Originator(发起人)：负责创建一个备忘录Memeto，用以记录当前时刻它的内部状态，并可以使用备忘录恢复内部状态。Originator可根据需要决定Memento储存Originator的哪些状态。

Memeto(备忘录)：负责存储Originator对象的内部状态，并可防止Originator意外的其它对象访问备忘录Memeto。备忘录有两个接口，Createaker只能看到备忘录的窄接口，它只能将备忘录传递给其它对象。Originator只能看到一个窄接口，允许它访问返回到先前状态所需的所有数据。

Caretaker(管理者)：负责保存好备忘录Memeto，不能对备忘录的内容进行操作或检查。

发起人(Originator)类

```
class Originator
{
    private string state;
    public string State
    {
        get {return state;}
        set {state = value;}
    }

    public Memnto CreateMemento()
    {
        return (new Memeto(state));
    }
    public void SetMemento(Memento memento)
    {
        state = memento.state
    }
    public void show()
    {
        Console.WriteLine("State="+ state);
    }
}
```

备忘录(Memento)类

```
class Memento
{
    private string state;
    public Memento(string state)
    {
        this.state = state;
    }
    public string State
    {
        get {return state;}
    }
}
```

管理者(Caretaker)类

```
class Caretaker
{
    private Memento memento;

    public Memento Memento
    {
        get {return memento;}
        set {memento = value;}
    }
}
```

客户端程序

```
static void Main(string[] args)
{
    Originator o = new Originator();
    o.State = "On";
    o.show();

    Caretaker c = new Caretaker();
    c.Memento = o.CreateMemento();

    o.State = "Off";
    o.Show();

    o.SetMemento(c.Memento);
    o.Show();

    Console.Read();

}
```

Memento模式比较适用于功能比较复杂的，但是需要维护或者纪录属性历史的类，或者需要保存的属性只是众多属性中的一小部分时，Originator可以根据保存的Memento信息还原到前一状态。

如果在某个系统中使用命令模式时，需要实现命令的撤销功能，那么命令模式可以使用备忘录模式来储存可撤销操作的状态。

当角色状态改变的时候，有可能这个状态无效，这时候就可以使用暂时存储起来的备忘录将状态复原。

游戏角色类

```
class 游戏角色
{
    ......

    //保存角色状态
    public RoleStateMemento SaveState()
    {
        return (new RoleStateMemento(vit, atk, def));
    }

    //恢复角色状态
    public void RecoveryState(RoleStateMemento memento)
    {
        this.vit = memento.Vitality;
        this.atk = memento.Attack;
        this.def = memento.Defense;
    }
    ......
}
```

角色状态储存箱类

```
class RoleStateMemento
{
    private int vit;
    private int atk;
    private int def;
    public RoleStateMemento(int vit, int atk, int def)
    {
        this.vit = vit;
        this.atk = atk;
        this.def = def;
    }
    //生命力
    public int Vtality
    {
        get {return vit;}
        set {vit = value;}
    }
    //攻击力
    public int Attack
    {
        get {return atk;}
        set {vit = value;}
    }
    //防御力
    public int Defense
    {
        get {return def;}
        set {def = value;}
    }
}
```

角色状态管理者

```
class RoleStateCaretaker
{
    private RoleStateMemeto memento;

    public RoleStateMemento Memento
    {
        get {return memento;}
        set {memento = value;}
    }
}
```

客户端代码

```
static void Main(string[] args)
{
    //大战Boss前
    GameRole lixiaoyao = new GameRole();
    lixiaoyao.GetInitState();
    lixiaoyao.StateDisplay();

    //保存进度
    RoleStateCaretaker stateAdmin = new RoleStateCaretaker();
    stateAdmin.memento = lixiaoyao.SaveState();

    //大战Boss时，损耗严重
    lixiaoyao.Fight();
    lixiaoyao.StateDisplay();

    //恢复之前状态
    lixiaoyao.RecoveryState(stateAdmin.Memento);
    lixiaoyao.StateDisplay();

    Console.Read();
}
```

组合模式(Composite Pattern)

简介

有个项目，是为一家在全国许多城市都有分销机构的大公司做办公管理系统，总部有人力资源，财务，运营等部门。但是总公司的人力资源部，财务部等办公管理功能在所有的分公司或办事处都需要有。我们可能希望人力资源部，财务部的管理功能可以复用于分公司。这其实就是整体与部分可以被一致对待的问题。

组合模式:将对象组合成树形结构以表示'部分-整体'的层次结构。组合模式使得用户对单个对象和组合对象的使用具有一致性。

当你发现需求中是体现部分与整体层次的结构时，以及你希望用户可以忽略组合对象与单个对象的不同，统一地使用组合结构中的所有对象时，就应该考虑使用组合模式了。

公司管理系统

公司类，抽象类或接口

```
abstract class Company
{
    protected string name;

    public Company(string name)
    {
        this.name = name;
    }
    public abstract void Add(Company c);//增加
    public abstract void Remove(Company c);//移除
    public abstract void Display(Company c);//显示    public abstra
}
```

具体公司类 实现接口树枝节点

```
class ConcreteCompany: Company
{
    private List<Company>children = new List<Company>();

    public ConcreteCompany(string name)
    {
        : base(name);
    }

    public override void Add(Company c)
    {
        children.Add(c);
    }
    public override void Remove(Company c)
    {
        children. Remove(c);
    }
    public override void Display(int depth)
    {
        Console.WriteLine(new string('-', depth) + name);

        foreach(Company component in children)
        {
            component.Display(depth + 2);
        }
    }

    public override void LineOfDuty()
    {
        foreach(Company component in children)
        {
            component.LineofDuty();
        }
    }
}
```

人力资源部与财务部类 树叶节点

```
class HRDepartment: Company
{
    public HRDepartment(string name)
    {
        : base(name);
    }

    public override void Add(Company c)
    {}

    public override void Remove(Company c)
    {}

    public override void Display(int depth)
    {
        Console.WriteLine(new string('-', depth) + name);
    }

    public override void LineOfDuty()
    {
        Console.WriteLine("{0} 员工招聘培训管理"，name);
    }
}
```

财务部

```
class FinanceDepartment: Company
{
    public FinanceDepartment(string name)
    {
        : base(name);
    }

    public override void Add(Company c)
    {}

    public override void Remove(Company c)
    {}

    public override void Display(int depth)
    {
        Console.WriteLine(new string('-', depth) + name);
    }

    public override void LineOfDuty()
    {
        Console.WriteLine("{0} 公司财务收支管理"，name);
    }
}
```

客户端调用

```
static void Main(string[] args)
{
    ConcreteCompany root = new ConcreteCompany("北京总公司");
    root.Add(new HRDepartment("总公司人力资源部"));
    root.Add(new FinanceDepartment("总公司财务部"));

    ConcreteCompany comp = new ConcreteCompany("上海华东分公司");
    comp.Add(new HRDepartment("总公司人力资源部"));
    comp.Add(new FinanceDepartment("总公司财务部"));
    root.Add(comp);

        ConcreteCompany comp1 = new ConcreteCompany("南京办事处");
    comp1.Add(new HRDepartment("南京办事处人力资源部"));
    comp1.Add(new FinanceDepartment("南京办事处财务部"));
    comp.Add(comp1);

    root.Display(1);
    root.LineOfDuty();
}
```

透明方式与安全方式

透明方式，也就是说在Component中声明所有用来管理子对象的方法，其中包括Add,Remove等。这样实现Component接口的所有子类都具备了Add和RFemove。这样做的好处就是叶节点和枝节点对于外界没有区别，它们具备完全一致的行为接口。但是问题也很明显，因为Leaf类本身不具备Add(),Remove()方法的功能，所以实现也是没意义的。

安全方式，也就是在Component接口中不去声明Add和Remove方法，那么字类的Leaf也不需要实现它，而是在Composite声明所有用来管理字类对象的方法。不过由于不透明，所以树枝类和树叶不具有相同的接口，客户端的调用需要做相应的判断，带来了不便。

组合模式有时候又叫做部分-整体模式，它使我们树型结构的问题中，模糊了简单元素和复杂元素的概念，客户程序可以向处理简单元素一样来处理复杂元素,从而使得客户程序与复杂元素的内部结构解耦。

单例模式(Singleton Pattern)

简介

单例模式，也叫单子模式，是一种常用的软件设计模式。在应用这个模式时，单例对象的类必须保证只有一个实例存在。许多时候整个系统只需要拥有一个的全局对象，这样有利于我们协调系统整体的行为。比如在某个服务器程序中，该服务器的配置信息存放在一个文件中，这些配置数据由一个单例对象统一读取，然后服务进程中的其他对象再通过这个单例对象获取这些配置信息。这种方式简化了在复杂环境下的配置管理。

实现单例模式的思路是：一个类能返回对象一个引用(永远是同一个)和一个获得该实例的方法（必须是静态方法，通常使用getInstance这个名称）；当我们调用这个方法时，如果类持有的引用不为空就返回这个引用，如果类保持的引用为空就创建该类的实例并将实例的引用赋予该类保持的引用；同时我们还将该类的构造函数定义为私有方法，这样其他处的代码就无法通过调用该类的构造函数来实例化该类的对象，只有通过该类提供的静态方法来得到该类的唯一实例。

单例模式在多线程的应用场合下必须小心使用。如果当唯一实例尚未创建时，有两个线程同时调用创建方法，那么它们同时没有检测到唯一实例的存在，从而同时各自创建了一个实例，这样就有两个实例被构造出来，从而违反了单例模式中实例唯一的原则。 解决这个问题的办法是为指示类是否已经实例化的变量提供一个互斥锁(虽然这样会降低效率)

JAVA的单例实现

```
public class Singleton {
  private static volatile Singleton INSTANCE = null;

  // Private constructor suppresses
  // default public constructor
  private Singleton() {}

  //thread safe and performance  promote
  public static  Singleton getInstance() {
      if(INSTANCE == null){
           synchronized(Singleton.class){
               //when more than two threads run into the first nu
               if(INSTANCE == null){
                   INSTANCE = new Singleton();
                }
            }
      }
      return INSTANCE;
  }
}
```

避免单例的滥用

单例模式固然好用，但感觉有点过度，将接口设计成单例入口前需要考虑一下：

- 这个类表达的含义真的只能有一个实例么？（如UIApplication）还是只是为了好调用而已？
- 这个单例持有的内存一直存在
- 是否能用类方法代替？
- 这个单例对象是否能成为另一个单例对象的属性？如果是，应该作为属性

实例

单例模式:保证一个类仅有一个实例，并提供一个访问它的全局访问点。

单例是整个 Cocoa 中被广泛使用的核心设计模式之一。事实上，苹果开发者库把单例作为 "Cocoa 核心竞争力" 之一。作为一个iOS开发者，我们经常和单例打交道，比如 UIApplication 和 NSFileManager 等等。我们在开源项目、苹果示例代码和 StackOverflow 中见过了无数使用单例的例子。Xcode 甚至有一个默认的 "Dispatch Once" 代码片段，可以使我们非常简单地在代码中添加一个单例：

```
+ (instancetype)sharedInstance
{
    static dispatch_once_t once;
    static id sharedInstance;
    dispatch_once(&once, ^{
        sharedInstance = [[self alloc] init];
    });
    return sharedInstance;
}
```

由于这些原因，单例在 iOS 开发中随处可见。问题是，它们很容易被滥用。

尽管有些人认为单例是 '反模式'，'魔鬼' 以及 '病态的说谎者'，我不会去完全否认单例所带来的的好处，而是会展示一些使用单例所带来的问题，这样下一次在使用 dispatch_once 代码片段的自动补全功能时，你可以对它的影响进行评估，三思而行。

全局状态

大多数的开发者都认同使用全局可变的状态是不好的行为。太多状态使得程序难以理解，难以调试。我们这些面向对象的程序员在最小化代码的状态复杂程度的方面，有很多需要向函数式编程学习的地方。

```
@implementation SPMath {
    NSUInteger _a;
    NSUInteger _b;
}

- (NSUInteger)computeSum
{
    return _a + _b;
}
```

在上面这个简单的数学库的实现中，程序员需要在调用 `computeSum` 前正确的设置实例变量 `_a` 和 `_b` 。这样有以下问题：

- computeSum 没有显式地通过使用参数的形式声明它依赖于 _a 和 _b 的状态。与仅仅通过查看函数声明就可以知道这个函数的输出依赖于哪些变量不同的是，另一个开发者必须查看这个函数的具体实现才能明白这个函数依赖那些变量。隐藏依赖是不好的。
- 当为调用 `computeSum` 做准备而修改 `_a` 和 `_b` 的数值时，程序员需要保证这些修改不会影响任何其他依赖于这两个变量的代码的正确性。而这在多线程的环境中是尤其困难的。

把下面的代码和上面的例子做对比：

```
+ (NSUInteger)computeSumOf:(NSUInteger)a plus:(NSUInteger)b
{
    return a + b;
}
```

这里，对变量 `a` 和 `b` 的依赖被显式地声明了。我们不需要为了调用这个方法而去改变实例变量的状态。并且我们也不需要担心调用这个函数会留下持久的副作用。我们甚至可以把这个方法声明为类方法，这样就告诉了代码的阅读者这个方法不会修改任何实例的状态。

那么，这个例子和单例又有什么关系呢？用 Miško Hevery 的话来说，单例就是披着羊皮的全局状态。一个单例可以被使用在任何地方，而不需要显式地声明依赖。就像变量 `_a` 和 `_b` 在 `computeSum` 内部被使用了，却没有被显式声明一样，程序的任意模块都可以调用 `[SPMySingleton sharedInstance]` 并且访问这个单例。这意味着任何和这个单例交互产生的副作用都会影响程序其他地方的任意代码。

```
@interface SPSingleton : NSObject

+ (instancetype)sharedInstance;

- (NSUInteger)badMutableState;
- (void)setBadMutableState:(NSUInteger)badMutableState;

@end

@implementation SPConsumerA

- (void)someMethod
{
    if ([[SPSingleton sharedInstance] badMutableState]) {
        // ...
    }
}

@end

@implementation SPConsumerB

- (void)someOtherMethod
{
    [[SPSingleton sharedInstance] setBadMutableState:0];
}

@end
```

在上面的例子中，`SPConsumerA` 和 `SPConsumerB` 是两个完全独立的模块。但是 `SPConsumerB` 可以通过使用单例提供的共享状态来影响 `SPConsumerA` 的行为。这种情况应该只能发生在 `consumer B` 显式引用了 A，并表明了两者之间的关系时。这里使用了单例，由于其具有全局和多状态的特性，导致隐式地在两个看起来完全不相关的模块之间建立了耦合。

让我们来看一个更具体的例子，并且暴露一个使用全局可变状态的额外问题。比如我们想要在我们的应用中构建一个网页查看器。为了支持这个查看器，我们构建了一个简单的 URL cache：

```
@interface SPURLCache

+ (SPCache *)sharedURLCache;

- (void)storeCachedResponse:(NSCachedURLResponse *)cachedResponse f

@end
```

这个开发者开始写一些单元测试来保证代码在一些不同的情况下都能达到预期。首先，他写了一个测试用例来保证网页查看器在设备没有连接时能够展示出错误信息。然后他写了一个测试用例来保证网页查看器能够正确的处理服务器错误。最后，他为成功情况时写了一个测试用例，来保证返回的网络内容能够被正确的显示出来。这个开发者运行了所有的测试用例，并且它们都如预期一样正确。赞！

几个月以后，这些测试用例开始出现失败，尽管网页查看器的代码从它写完后就从来没有再改动过！到底发生了什么？

原来，有人改变了测试的顺序。处理成功的那个测试用例首先被运行，然后再运行其他两个。处理错误的那两个测试用例现在竟然成功了，和预期不一样，因为 URL cache 这个单例把不同测试用例之间的 response 缓存起来了。

持久化状态是单元测试的敌人，因为单元测试在各个测试用例相互独立的情况下才有效。如果状态从一个测试用例传递到了另外一个，这样就和测试用例的执行顺序就有关系了。有 bug 的测试用例，尤其是那些本来不应该通过的测试用例，是非常糟糕的事情。

对象的生命周期

另外一个关键问题就是单例的生命周期。当你在程序中添加一个单例时，很容易会认为“永远只会有一个实例”。但是在很多我看到过的 iOS 代码中，这种假定都可能被打破。

比如，假设我们正在构建一个应用，在这个应用里用户可以看到他们的好友列表。他们的每个朋友都有一张个人信息的图片，并且我们想使我们的应用能够下载并且在设备上缓存这些图片。 使用 dispatch_once 代码片段，我们可以写一个 SPThumbnailCache 单例：

```
@interface SPThumbnailCache : NSObject

+ (instancetype)sharedThumbnailCache;

- (void)cacheProfileImage:(NSData *)imageData forUserId:(NSString *
- (NSData *)cachedProfileImageForUserId:(NSString *)userId;

@end
```

我们继续构建我们的应用，一切看起来都很正常，直到有一天，我们决定去实现‘注销’功能，这样用户可以在应用中进行账号切换。突然我们发现我们将要面临一个讨厌的问题：用户相关的状态存储在全局单例中。当用户注销后，我们希望能够清理掉所有的硬盘上的持久化状态。否则，我们将会把这些被遗弃的数据残留在用户的设备上，浪费宝贵的硬盘空间。对于用户登出又登录了一个新的账号这种情况，我们也想能够对这个新用户使用一个全新的 SPThumbnailCache 实例。

问题在于按照定义单例被认为是“创建一次，永久有效”的实例。你可以想到一些对于上述问题的解决方案。或许我们可以在用户登出时移除这个单例：

```
static SPThumbnailCache *sharedThumbnailCache;

+ (instancetype)sharedThumbnailCache
{
    if (!sharedThumbnailCache) {
        sharedThumbnailCache = [[self alloc] init];
    }
    return sharedThumbnailCache;
}

+ (void)tearDown
{
    // The SPThumbnailCache will clean up persistent states when de
    sharedThumbnailCache = nil;
}
```

这是一个明显的对单例模式的滥用，但是它可以工作，对吧？

我们当然可以使用这种方式去解决，但是代价实在是太大了。我们不能使用简单的的 `dispatch_once` 方案了，而这个方案能够保证线程安全以及所有调用 `[SPThumbnailCache sharedThumbnailCache]` 的地方都能访问到同一个实例。现在我们需要对使用缩略图 cache 的代码的执行顺序非常小心。假设当用户正在执行登出操作时，有一些后台任务正在执行把图片保存到缓存中的操作:

```
dispatch_async(dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DE
    [[SPThumbnailCache sharedThumbnailCache] cacheProfileImage:newI
});
```

我们需要保证在所有的后台任务完成前， tearDown 一定不能被执行。这确保了 newImage 数据可以被正确的清理掉。或者，我们需要保证在缩略图 cache 被移除时，后台缓存任务一定要被取消掉。否则，一个新的缩略图 cache 的实例将会被延迟创建，并且之前用户的数据 (newImage 对象) 会被存储在它里面。

由于对于单例实例来说它没有明确的所有者，(因为单例自己管理自己的生命周期)，“关闭”一个单例变得非常的困难。

分析到这里，我希望你能够意识到，“这个缩略图 cache 从来就不应该作为一个单例！”。问题在于一个对象得生命周期可能在项目的最初阶段没有被很好得考虑清楚。举一个具体的例子，Dropbox 的 iOS 客户端曾经只支持一个账号登录。它以这样的状态存在了数年，直到有一天我们希望能够同时支持多个用户账号登录 (同时登陆私人账号和工作账号)。突然之间，我们以前的的假设“只能够同时有一个用户处于登录状态”就不成立了。如果假定了一个对象的生命周期和应用的生命周期一致，那你的代码的灵活扩展就受到了限制，早晚有一天当产品的需求产生变化时，你会为当初的这个假定付出代价的。

这里我们得到的教训是，单例应该只用来保存全局的状态，并且不能和任何作用域绑定。如果这些状态的作用域比一个完整的应用程序的生命周期要短，那么这个状态就不应该使用单例来管理。用一个单例来管理用户绑定的状态，是代码的坏味道，你应该认真的重新评估你的对象图的设计。

避免使用单例

既然单例对局部作用域的状态有这么多的坏处，那么我们应该怎样避免使用它们呢？

让我们来重温一下上面的例子。既然我们的缩略图 cache 的缓存状态是和具体的用户绑定的，那么让我们来定义一个user对象吧：

```
@interface SPUser : NSObject

@property (nonatomic, readonly) SPThumbnailCache *thumbnailCache;

@end

@implementation SPUser

- (instancetype)init
{
    if ((self = [super init])) {
        _thumbnailCache = [[SPThumbnailCache alloc] init];

        // Initialize other user-specific state...
    }
    return self;
}

@end
```

我们现在用一个对象来作为一个经过认证的用户会话的模型类，并且我们可以把所有和用户相关的状态存储在这个对象中。现在假设我们有一个view controller来展现好友列表：

```
@interface SPFriendListViewController : UIViewController

- (instancetype)initWithUser:(SPUser *)user;

@end
```

我们可以显式地把经过认证的 user 对象作为参数传递给这个 view controller。这种把依赖性传递给依赖对象的技术正式的叫法是依赖注入，它有很多优点：

1. 对于阅读这个 SPFriendListViewController 头文件的读者来说，可以很清楚的知道它只有在有登录用户的情况下才会被展示。
2. 这个 SPFriendListViewController 只要还在使用中，就可以强引用 user 对象。举例来说，对于前面的例子，我们可以像下面这样在后台任务中保存一个图片到缩略图 cache 中：

```
dispatch_async(dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DE
    [_user.thumbnailCache cacheProfileImage:newImage forUserId:use
});
```

就算后台任务还没有完成，应用其他地方的代码也可以创建和使用一个全新的 SPUser 对象，而不会在清理第一个实例时阻塞用户交互。

为了更详细的说明一下第二点，让我们画一下在使用依赖注入之前和之后的对象图。

假设我们的 SPFriendListViewController 是当前 window 的 root view controller。使用单例时，我们的对象图看起来如下所示：



view controller 自己，以及自定义的 image view 的列表，都会和 sharedThumbnailCache 产生交互。当用户登出后，我们想要清理 root view controller 并且退出到登录页面：



这里的问题在于这个好友列表的 view controller 可能仍然在执行代码 (由于后台操作的原因)，并且可能因此仍然有一些没有执行的涉及到 sharedThumbnailCache 的调用。

和使用依赖注入的解决方案对比一下：



简单起见，假设 SPApplicationDelegate 管理 SPUser 的实例 (在实践中，你可能会把这些用户状态的管理工作交给另外一个对象来做，这样可以使你的 application delegate 简化)。当展现好友列表 view controller 时，会传递进去一个 user 的引用。这个引用也会向下传递给 profile image views。现在，当用户登出时，我们的对象图如下所示：



这个对象图看起来和使用单例时很像。那么，区别是什么呢？

关键问题是作用域。在单例那种情况中，sharedThumbnailCache 仍然可以被程序的任意模块访问。假如用户快速的登录了一个新的账号。该用户也想看看他的好友列表，这也就意味着需要再一次的和缩略图 cache 产生交互：



当用户登录一个新账号，我们应该能够构建并且与全新的 SPThumbnailCache 交互，而不需要再在销毁老的缩略图 cache 上花费精力。基于对象管理的典型规则，老的 view controllers 和老的缩略图 cache 应该能够自己在后台延迟被清理掉。简而言之，我们应该隔离用户 A 相关联的状态和用户 B 相关联的状态：



这一切的关键点是，在面向对象编程中我们想要最小化可变状态的作用域。但是单例却因为使可变的状态可以被程序中的任何地方访问，而站在了对立面。下一次你想使用单例时，我希望你能够好好考虑一下使用依赖注入作为替代方案。

桥接模式(Bridge Pattern)

简介

桥接模式把事物对象和其具体行为、具体特征分离开来，使它们可以各自独立的变化。事物对象仅是一个抽象的概念。如“圆形”、“三角形”归于抽象的“形状”之下，而“画圆”、“画三角”归于实现行为的“画图”类之下，然后由“形状”调用“画图”。

如果有一个N品牌的手机，它有个小游戏，我要玩游戏，程序应该如何写？

```
//N品牌的手机中的游戏
class HandSetNGame
{
    public void Run()
    {
        Console.WriteLine("运行N品牌");
    }
}
```

客户端

```
HandSetNGame game = new HandSetGame();
game.run();
```

现在又有一个M品牌的手机，也是小游戏，客户端也可以调用，如何做？

```
class HandSetGame
{
    public virtual void Run()
    {
    }
}
```

M品牌手机游戏和N品牌手机游戏

```
class HandSetMGame : HandSetGanme
{
    public override void Run()
    {
        Console.WriteLine("运行M品牌手机游戏");
    }
}

class HandSetNGame : HandSetGanme
{
    public override void Run()
    {
        Console.WriteLine("运行N品牌手机游戏");
    }
}
```

然后，由于手机都需要通讯录功能，于是N品牌和M品牌都增加了通讯录的增删该查功能，如何处理？

```
//手机品牌
class HandSetBrand
{
    public virtual void run()
    {
    }
}
```

```
//手机品牌M
class HandSetBrandM:HandsetBrand
{
}

//手机品牌N
class HandSetBrandM:HandsetBrand
{
}
```

下属的各自通讯录类和游戏类

```
//手机品牌M的游戏
class HandSetBrandMGame:HandSetBrandM
{
    public override void Run()
    {
        Console.WriteLine("运行M品牌手机游戏");
    }
}

//手机品牌N的游戏
class HandSetBrandMGame:HandSetBrandM
{
    public override void Run()
    {
        Console.WriteLine("运行N品牌手机游戏");
    }
}

//手机品牌M的通讯录
class HandSetBrandMAddressList:HandSetBrandM
{
    public override void Run()
    {
        Console.WriteLine("运行M品牌手机通讯录");
    }
}

//手机品牌N的
class HandSetBrandMAddressList:HandSetBrandM
{
    public override void Run()
    {
        Console.WriteLine("运行N品牌手机通讯录");
    }
}
```

客户端调用代码

```
static void Main(strring[] args)
{
    HandSetBrand ab;

    ab = new HandSetBrandMAddressList();
    ab.Run();

    ab = new HandSetBrandNAddressList();
    ab.Run();

    ab = new HandSetBrandMGame();
    ab.Run();

    ab = new HandSetBrandNGame();
    ab.Run();

    Console.Read();
}
```

如果我现在需要每个品牌都增加一个MP3音乐功能，如何做？还增加子类？

对象的继承关系是在编译时就定义好的，所以无法在运行时改变从父类继承的实现，子类的实现与它的父类有非常紧密的依赖关系，以至于父类实现中的任何变化必然会导致子类发生变化。当你需要复用子类时，如果继承下来的实现不适合解决新的问题，则父类必须重写或被其它更适合的类替换。这种依赖关系限制了灵活性并最终限制了复用性。

合成***(Composition)***／聚合***(Aggregation)***复用原则**,**尽量使用合成／聚合，尽量不要使用类继承。

聚合表示一种弱的'拥有'关系，体现的是A对象可以包含B对象，但是B对象不是A对象的一部分。合成则是一种强的'拥有'关系，体现了严格的部分和整体的关系，部分和整体的生命周期一样。

比如说 大雁有翅膀，大雁和翅膀是部分和整体的关系，并且它们生命周期是一样的，于是它们就是合成关系。而大雁是群居动物，所以每只大雁都属于一个雁群，一个雁群可以有很多大雁，所以大雁和雁群是聚合关系。

优先使用对象的合成／聚合将有助于你保持每个类被封装，并被集中在单个任务上。这样类和类继承层次会保持较小规模，并且不太可能增长为不可控制的庞然大物。

像'游戏'，'通讯录'，'MP3音乐'，如果我们可以让其分离与手机的耦合，那么可以大大减少面对新需求改动过大的不合理情况。

松耦合的程序

```
//手机软件
abstract class HandsetSoft
{
    public abstract void Run();
}
```

游戏，通讯录等具体类

```
//手机游戏
class HandsetGame : HandSetSoft
{
    public override void Run()
    {
        Console.WriteLine("运行手机游戏");
    }
}

//手机通讯录
class HandsetAddressList : HandSetSoft
{
    public override void Run()
    {
        Console.WriteLine("运行手机通讯录");
    }
}
```

```
//手机品牌
abstract class HandsetBrand
{
    protected HandsetSoft soft;

    //设置手机软件
    public void SetHandSetSoft(HandSetSoft soft)
    {
        this.soft = soft;
    }
    //运行
    public abtract void run();
}
```

品牌N品牌M具体类

```
//手机品牌N
class HandSetBrandN: HandBrand
{
    public override void Run()
    {
        soft.Run();
    }
}

//手机品牌M
class HandSetBrandM: HandBrand
{
    public override void Run()
    {
        soft.Run();
    }
}
```

客户端代码

```
static void Mian(string[] args)
{
    HandSetBrand ab;
    ab = new HandSetBrandN();

    ab.setHandSetSoft(new HandSetGame());
    ab.run();

    ab.setHandSetSoft(new HandSetAddressList());
    ab.run();

    ab = new HandSetBrandM();

    ab.setHandSetSoft(new HandSetGame());
    ab.run();

    ab.setHandSetSoft(new HandSetAddressList());
    ab.run();

    Console.Read();
}
```

现在我们要再加个功能 比如说MP3功能，只要增加这个类就好了，不会影响其它任何类。

```
//手机MP3播放
class HandsetMP3 : HandSetSoft
{
    public override void Run()
    {
        Console.WriteLine("运行手机MP3");
    }
}
```

如果要增加S品牌，只需要增加一个品牌子类就可以了。

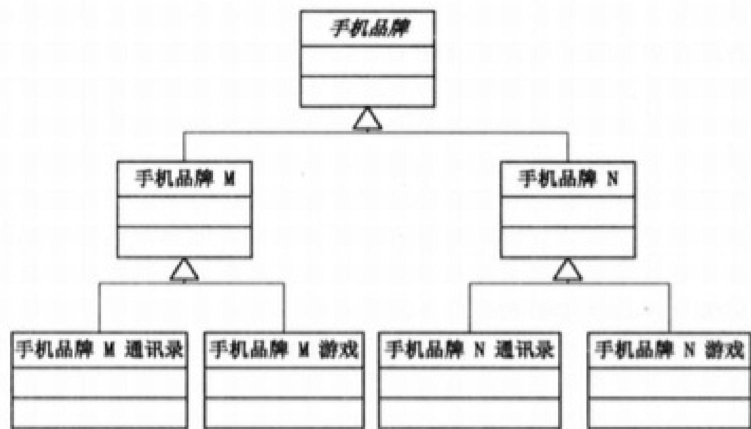
```
//手机品牌S
class HandSetBrand: HandSetBrand
{
    public override void Run()
    {
        soft.Run();
    }
}
```

桥接模式

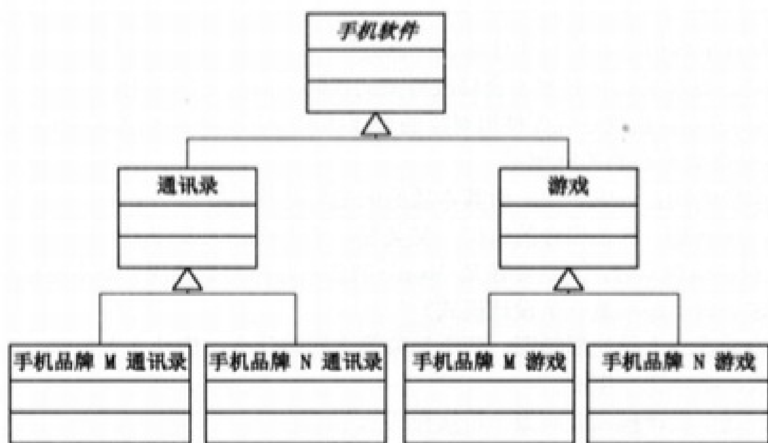
桥接模式:将抽象部分与它的实现部分分离，使它们都可以独立地变化。

什么叫抽象与它的实现分离，这并不是说，让抽象与其派生类分离，因为这没有任何意义。实现指的是抽象类和它的派生类用来实现自己的对象。

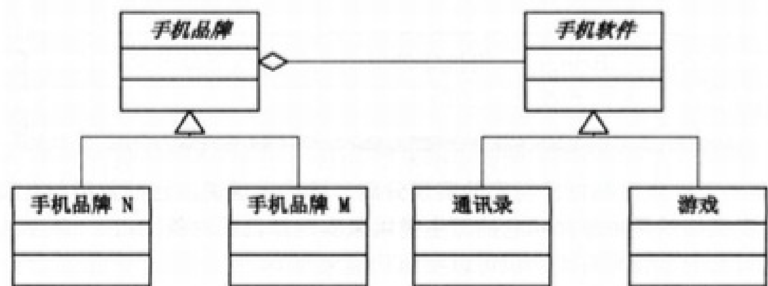
按品牌分类实现结构图



按软件分类实现结构图



"由于实现的方式有多种，桥接模式的核心意图就是把这些实现独立出来，让它们各自地变化。这就使得每种实现的变化不会影响其他实现，从而达到应对变化的目的。"



桥接模式基本代码



桥接模式把两个角色之间的继承关系改为聚合关系，从而使二者可以各自独立的变化。把原来在基类的实现化细节抽象出来，再构造到一个实现化的结构中，然后把原来的基类改造成一个抽象化的等级结构，这样就实现了系统在多个维度上的独立变化。

命令模式(Command Pattern)

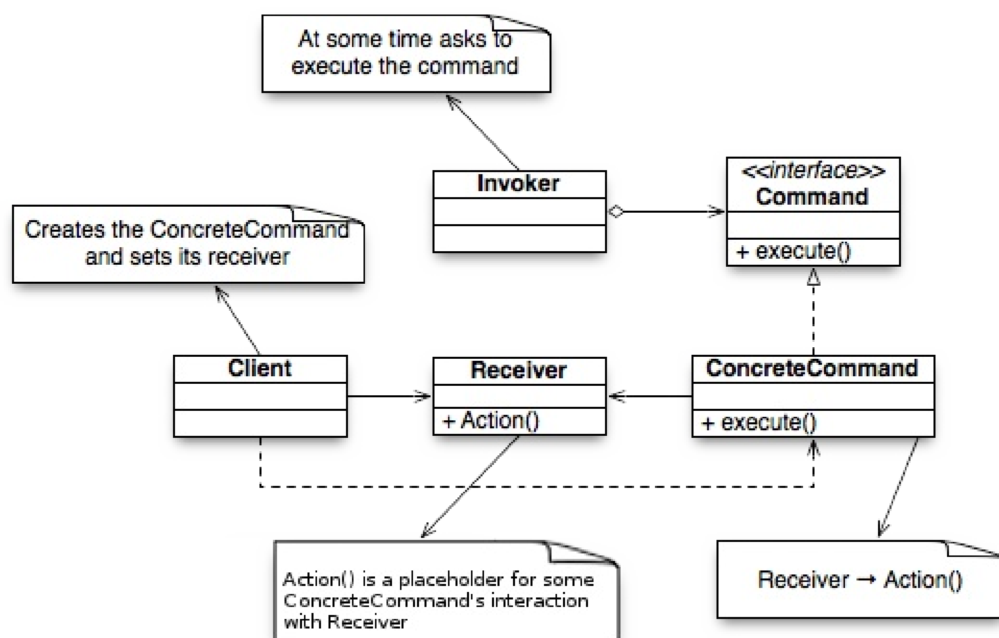
简介

在面向对象程式设计的范畴中，命令模式是一种设计模式，它尝试以物件来代表实际行动。命令物件可以把行动(action) 及其参数封装起来，于是这些行动可以被：

- 重复多次
- 取消（如果该物件有实作的话）
- 取消后又再重做

这些都是现代大型应用程序所必须的功能，即“复原”及“重复”。除此之外，可以用命令模式来实作的功能例子还有：

- 交易行为
- 进度列
- 精灵
- 使用者界面按钮及功能表项目
- 执行绪 pool
- 宏收录



JavaScript

```
/* The Invoker function */
var Switch = function(){
    var _commands = [];
    this.storeAndExecute = function(command){
        _commands.push(command);
        command.execute();
    }
}

/* The Receiver function */
var Light = function(){
    this.turnOn = function(){ console.log ('turn on')};
    this.turnOff = function(){ console.log ('turn off') };
}

/* The Command for turning on the light - ConcreteCommand #1 */
var FlipUpCommand = function(light){
    this.execute = light.turnOn;
}

/* The Command for turning off the light - ConcreteCommand #2 */
var FlipDownCommand = function(light){
    this.execute = light.turnOff;
}

var light = new Light();
var switchUp = new FlipUpCommand(light);
var switchDown = new FlipDownCommand(light);
var s = new Switch();

s.storeAndExecute(switchUp);
s.storeAndExecute(switchDown);
```

Java

```
import java.util.List;
import java.util.ArrayList;

/* The Command interface */
public interface Command {
   void execute();
}

/* The Invoker class */
public class Switch {
   private List<Command> history = new ArrayList<Command>();

   public Switch() {
   }

   public void storeAndExecute(Command cmd) {
```

```
        this.history.add(cmd); // optional
        cmd.execute();
    }
}

/* The Receiver class */
public class Light {
    public Light() {
    }

    public void turnOn() {
        System.out.println("The light is on");
    }

    public void turnOff() {
        System.out.println("The light is off");
    }
}

/* The Command for turning on the light - ConcreteCommand #1 */
public class FlipUpCommand implements Command {
    private Light theLight;

    public FlipUpCommand(Light light) {
        this.theLight = light;
    }

    public void execute(){
        theLight.turnOn();
    }
}

/* The Command for turning off the light - ConcreteCommand #2 */
public class FlipDownCommand implements Command {
    private Light theLight;

    public FlipDownCommand(Light light) {
        this.theLight = light;
    }

    public void execute() {
        theLight.turnOff();
    }
}

/* The test class or client */
public class PressSwitch {
    public static void main(String[] args){
        Light lamp = new Light();
        Command switchUp = new FlipUpCommand(lamp);
        Command switchDown = new FlipDownCommand(lamp);

        Switch mySwitch = new Switch();
```

```
        try {
            if ("ON".equalsIgnoreCase(args[0])) {
                mySwitch.storeAndExecute(switchUp);
            }
            else if ("OFF".equalsIgnoreCase(args[0])) {
                mySwitch.storeAndExecute(switchDown);
            }
            else {
                System.out.println("Argument \"ON\" or \"OFF\" is requi
            }
        } catch (Exception e) {
            System.out.println("Arguments required.");
        }
    }
}
```

实例

模拟烧烤

紧耦合设计

```
//烤肉串者
public class Barbecuer
{
    //烤羊肉
    public void BakeMutton()
    {
        Console.WriteMutton("烤羊肉串");
    }
    //烤鸡翅
    public void BakeChikenWing()
    {
        Console.WriteMutton("烤鸡翅");
    }

}
```

客户端调用

```
static void Main(string[] args)
{
    Barbecuer boy = new Barbecuew();
    boy.BakeMutton();
    boy.BakeMutton();
    boy.BakeMutton();
    boy.BakeChickenWing();
}
```

松耦合设计

抽象命令类

```
//抽象命令
public abstract class Command
{
    protected Barbecuer receiver;

    public Command(Barbecuer receiver)
    {
        this.receiver = receiver;
    }

    //执行命令
    abstract public ExcuteCommand();
}
```

具体命令类

```
//烤羊肉命令
class BakeMuttonCommand: Command
{
    public BakeMuttonCommand(Barbecuer receiver)
    {
        :base(receiver)
    }

    public override void EccuteCommand()
    {
        receiver.BakeMutton();
    }
}

//烤鸡翅命令
class BakeMuttonCommand: Command
{
    public BakeMuttonCommand(Barbecuer receiver)
    {
        :base(receiver)
    }

    public override void EccuteCommand()
    {
        receiver.BakeChickenWing();
    }
}
```

服务员类

```
//服务员
public class Waiter
{
    private Command command;

    //设置订单
    public void setOrder(Command command)
    {
        this.command
    }

    //通知执行
    public void Notify()
    {
        command.ExcuteCommand();
    }
}
```

```
//烤肉串者
public class Barbecuer
{
    //烤羊肉
    public void BakeMutton()
    {
        Console.WriteMutton("烤羊肉串");
    }
    //烤鸡翅
    public void BakeChikenWing()
    {
        Console.WriteMutton("烤鸡翅");
    }

}
```

客户端实现

```
static void Main(string[] args)
{
    //开店前的准备
    Barbecuer boy = new Barbecuer();
    Command BakeMuttonCommand1 = new BakeMuttonCommand(boy);
    Command BakeMuttonCommand2 = new BakeMuttonCommand(boy);
    Command BakeChickenWingCommand1 = new BakeMuttonCommand(boy);

    Waiter girl = new Waiter();

    //开门营业
    girl.SetOrder(BakeMuttonCommand1);
    girl.Notify();

    girl.SetOrder(BakeMuttonCommand2);
    girl.Notify();

    girl.SetOrder(BakeChickenWingCommand1);
    girl.Notify();

    COnsole.Read();
}
```

松耦合后

```
//服务员
public class Waiter
{
    private IList<Command> orders = new List<Command>();

    //设置订单
    public void SetOrder(Command command)
    {
        if (command.ToString() == "命令模式.BakeChikenWingCommand")
        {
            Console.WriteLine("服务员：鸡翅没有了，请点别的烧烤。");
        }
        else
        {
            orders.Add(command);
            Console.WriteLine("增加订单："+ command.ToString() + "时
        }
    }

    //取消订单
    public void CancelOrder(Command command)
    {
        orders.Remove(command);
        Console.WriteLine("取消订单" + command.ToString() + "时间" +
    }

    //通知全部执行
    public void Notify()
    {
        foreach(Command cmd in orders)
        {
            cmd.ExcuteCommand();
        }
    }
}
```

客户端代码实现

```
static void Main(string[] args)
{

    Barbecuer boy = new Barbecuer();
    Command BakeMuttonCommand1 = new BakeMuttonCommand(boy);
    Command BakeMuttonCommand2 = new BakeMuttonCommand(boy);
    Command BakeChickenWingCommand1 = new BakeMuttonCommand(boy);

    Waiter girl = new Waiter();

    girl.setOrder(bakeMuttonCommand1);
    girl.setOrder(bakeMuttonCommand1);
    girl.setOrder(BakeChickenWingCommand1);

    girl.Notify();

    Console.Read();
}
```

命令模式**:**将一个请求封装为一个对象，从而使你可用不同的请求对客户进行参数话；对请求排队或者纪录请求日志，以及支持可撤销的操作。

命令模式的优点:

- 容易地设计一个命令队列。
- 在需求的情况下，比较容易地将命令记入日志。
- 允许接受请求的一方决定是否要回绝请求。
- 很容易对请求撤销或者重做。
- 加入新的命令类不影响其它的类。
- 把请求一个操作的对象与知道怎么执行一个操作的对象分割开。

责任链模式(Chain-of-responsibility pattern)

简介

责任链模式在面向对象程式设计里是一种软件设计模式，它包含了一些命令对象和一系列的处理对象。每一个处理对象决定它能处理哪些命令对象，它也知道如何将它不能处理的命令对象传递给该链中的下一个处理对象。该模式还描述了往该处理链的末尾添加新的处理对象的方法。

以下的日志类(logging)例子演示了该模式。 每一个logging handler首先决定是否需要在该层做处理，然后将控制传递到下一个logging handler。程序的输出是:

```
  Writing to debug output: Entering function y.
  Writing to debug output: Step1 completed.
  Sending via e-mail:      Step1 completed.
  Writing to debug output: An error has occurred.
  Sending via e-mail:      An error has occurred.
  Writing to stderr:       An error has occurred.
```

注意：该例子不是日志类的推荐实现方式。

同时，需要注意的是，通常在责任链模式的实现中，如果在某一层已经处理了这个logger，那么这个logger就不会传递下去。在我们这个例子中，消息会一直传递到最底层不管它是否已经被处理。

```
import java.util.*;

abstract class Logger
{
    public static int ERR = 3;
    public static int NOTICE = 5;
    public static int DEBUG = 7;
    protected int mask;

    // The next element in the chain of responsibility
    protected Logger next;
    public Logger setNext( Logger l)
    {
        next = l;
        return this;
    }

    public final void message( String msg, int priority )
    {
        if ( priority <= mask )
        {
```

```
            writeMessage( msg );
            if ( next != null )
            {
                next.message( msg, priority );
            }
        }
    }

    protected abstract void writeMessage( String msg );

}

class StdoutLogger extends Logger
{

    public StdoutLogger( int mask ) { this.mask = mask; }

    protected void writeMessage( String msg )
    {
        System.out.println( "Writting to stdout: " + msg );
    }
}


class EmailLogger extends Logger
{

    public EmailLogger( int mask ) { this.mask = mask; }

    protected void writeMessage( String msg )
    {
        System.out.println( "Sending via email: " + msg );
    }
}

class StderrLogger extends Logger
{

    public StderrLogger( int mask ) { this.mask = mask; }

    protected void writeMessage( String msg )
    {
        System.out.println( "Sending to stderr: " + msg );
    }
}

public class ChainOfResponsibilityExample
{
    public static void main( String[] args )
    {
        // Build the chain of responsibility
        Logger l = new StdoutLogger( Logger.DEBUG).setNext(
                            new EmailLogger( Logger.NOTICE ).setNe
```

```
                            new StderrLogger( Logger.ERR ) ) );

        // Handled by StdoutLogger
        l.message( "Entering function y.", Logger.DEBUG );

        // Handled by StdoutLogger and EmailLogger
        l.message( "Step1 completed.", Logger.NOTICE );

        // Handled by all three loggers
        l.message( "An error has occurred.", Logger.ERR );
    }
}
```

加薪代码初步

```
//申请
class Request
{
    //申请类别
    private string requestType;
    public string RequestType
    {
        get {return requestType;}
        set {requestType = value;}
    }

    //申请内容
    private string requestContent;
    public string RequestContent
    {
        get {return RequestContent;}
        set {RequestContent = value;}
    }

    //数量
    private int number;
    public int Number
    {
        get {return number;}
        set {number = value;}
    }
}
```

```
//管理者
class Manager
{
    protected string name;
    public Manager(string name)
    {
        this.name = name;
    }

    //得到结果
    public void GetResult(string managerLevel, Request request)
    {
        if (request.RequestType=="经理") {
            if (request.RequestType == "请假" && request.Number <=
            {
                Console.WriteLine("被批准");
            } else
            {
                Console.WriteLine("我无权处理");
            }
        } else if (request.RequestType=="总监") {
            if (request.RequestType == "请假" && request.Number <=
            {
                Console.WriteLine("被批准");
            } else
            {
                Console.WriteLine("我无权处理");
            }
        } else if (request.RequestType=="总经理") {
            if (request.RequestType == "请假“) {
                Console.WriteLine("被批准");
            } else if (request.RequestType == "加薪" && request.Num
                Console.WriteLine("被批准");
            } else if (request.RequestType == "加薪" && request.Num
                Console.WriteLine("再说");
            }
        }
    }
}
```

客户端代码如下

```
static void Main(string[] args)
{
    Manager jinli = new Manager("金利");
    Manager zongjian = new Manager("宗剑");
    Manager zhongjingli = new Manager("钟精励");

    Request request = new Reuqest();
    request.RequestType = "加薪";
    request.RequestContent = "小菜请求加薪";
    request.Number = 1000;

    Request request2 = new Reuqest();
    request2.RequestType = "请假";
    request2.RequestContent = "小菜请假";
    request2.Number = 3;

    jinli.GetResult("经理",request2);
    zongjian.GetResult("总监",request2);
    zhongjingli.GetResult("总经理",request2);

    Console.Read();
}
```

代码评判:'管理者'类里面的'结果'方法比较长，加上有太多的分支判断，这种设计很不好。而且会不会增加其它的管理类别，比如说项目经理，部门经理，那就意味者需要去更改这个类，这个类承担了太多的责任，这违背了哪些设计原则？

类有太多的责任，这违背了单一职责原则，增加新的管理类别，需要修改这个类，违背了开放—封闭原则。

子类化加多太改善。

职责链模式:使多个对象都有机会处理请求，从而避免请求的发送者和接受者之间的耦合关系，将这个对象连成一条链，并沿着个链传递请求，直到有一个对象处理为止。

加薪代码重构

```
//管理者
abstract class Manager
{
    protected Manager superior;

    public Manager(string name)
    {
        this.name = name;
    }

    //设置管理者的上级
    public void setSuperior(Manager superior)
    {
        this.superior = superior;
    }
}
```

经理类就可以去继承这个'管理者'类，只需重写'申请要求'的方法就可以了。

```
class CommonManager : Manager
{
    public CommonManager(string name) :base(name)
    {}

    public override void RequestApplications(Request request)
    {
        if (request.RequestType == "请假" && request.Number <= 2)
        {
            Console.WriteLine("被批准");
        } else
        {
            if (superior != null) superior.RequestApplications(requ
        }
    }
}
```

总监类同样继承'管理者类'。

```
//总监
class Majordomo :Manager
{
    public Major(string name): base(name)
    {}
    public override void RequestApplications(Request request)
    {
        if (request.RequestType == "请假" && request.Number <= 2)
        {
            Console.WriteLine("被批准");
        } else
        {
            if (superior != null) superior.RequestApplications(requ
        }
    }
}
```

总经理类的权限就是全部都需要处理。

```
//总经理
class GeneralManager :Manager
{
    public GeneralManager(string name): base(name)
    {}
    public override void RequestApplications(Request request)
    {
        if (request.RequestType == "请假")
        {
            Console.WriteLine("被批准");
        } else if (request.RequestType == "加薪" && request.Number
        {
            Console.WriteLine("被批准");
        } else if (request.RequestType == "加薪" && request.Number
        {
            Console.WriteLine("再说吧");
        }
    }
}
```

由于我们把你原来的一个'管理者'类改成了一个抽象类和三个具体类，此时类之间的灵活性就大大的增加了，如果我们需要扩展新的管理类别，只需要增加子类就可以。当然，还有一个关键，那就是客户端如何编写。

```
static void Main(string[] args)
{
    CommonManager jinli = new CommonManger("金利");
    Majordomo zongjian = new Majordomo("宗建");
    GeneralManager zhongjingli = newGeneralManager("钟精利");

    //设置上级
    jinli.SetSuperior(zongjian);
    zongjian.SetSuperior(zhongjingli);

    Request request = new Request();
    request.RequestType = "请假";
    request.RequestContent = "小菜请假";
    request.Number = 1;
    //客户端的申请都是有‘经理’发起，但是实际谁来决策都是由具体管理类处理，客户
    jinli.requestApplications(request);

    Request request1 = new Request();
    request1.RequestType = "请假";
    request1.RequestContent = "小菜请假";
    request1.Number = 4;
    jinli.requestApplications(request1);

    Request request2 = new Request();
    request2.RequestType = "请求加薪";
    request2.RequestContent = "小菜请加薪";
    request2.Number = 400;
    jinli.requestApplications(request2);

    Console.Read();
}
```

责任链模式是一种对象的行为模式【GOF95】。在责任链模式里，很多对象由每一个对象对其下家的引用而连接起来形成一条链。请求在这个链上传递，直到链上的某一个对象决定处理此请求。发出这个请求的客户端并不知道链上的哪一个对象最终处理这个请求，这使得系统可以在不影响客户端的情况下动态地重新组织链和分配责任。

门框夹核桃

职责链模式想要做到的事情其实就是把多个函数链起来调用。

该模式提出的时候FP并不如今日盛行，其作者选用类来包装需要被链接的多个函数，这无可厚非。

无论是class，还是function，都是为程序员提供抽象的手段。当我们想要链接的东西就是多个function，选择直接用function而非class就会显得更加自然，也更加轻量且合适。

```
object Loggers {
  val ERR = 3
  val NOTICE = 5
  val DEBUG = 7

  case class Event(message: String, priority: Int)

  type Logger = Event => Event

  def stdOutLogger(mask: Int): Logger = event => handleEvent(event,
    println(s"Writing to stdout: ${event.message}")
  }

  def emailLogger(mask: Int): Logger = event => handleEvent(event, 
    println(s"Sending via e-mail: ${event.message}")
  }

  def stdErrLogger(mask: Int): Logger = event => handleEvent(event,
    System.err.println(s"Sending to stderr: ${event.message}")
  }

  private def handleEvent(event: Event, mask: Int)(handler: => Unit
    if (event.priority <= mask) handler
    event
  }
}
```

三个log的的等级ERR，NOTICE和DEBUG和之前Java的实现是一样的。

一个case class Event，用来包裹需要被log的事件。

type Logger则是声明了一个函数签名，凡是符合这个签名的函数都可以作为logger被使用。

然后便是三个函数实现，它们将mask通过闭包封进函数内。这三个函数共同依赖一个私有handleEvent函数，其作用和Java代码中的message类似，判断mask和正在发生的事件之间优先级大小关系，并以此决定当前logger是否需要处理该事件。

哎？等一下，这个是职责链模式啊，那个啥，链在哪儿呢？

```
object ChainRunner {

  import chain.Loggers._

  def main(args: Array[String]) {
    val chain = stdOutLogger(DEBUG) andThen emailLogger(NOTICE) anc

    chain(Event("Entering function y.", DEBUG))
    chain(Event("Step1 completed.", NOTICE))
    chain(Event("An error has occurred.", ERR))
  }
}
```

以上代码中的andThen就可以把三个logger链在一起。

中介者模式(Mediator Pattern)

简介

在用户与用户直接聊天的设计方案中，用户对象之间存在很强的关联性，将导致系统出现如下问题：

- 系统结构复杂：对象之间存在大量的相互关联和调用，若有一个对象发生变化，则需要跟踪和该对象关联的其他所有对象，并进行适当处理。
- 对象可重用性差：由于一个对象和其他对象具有很强的关联，若没有其他对象的支持，一个对象很难被另一个系统或模块重用，这些对象表现出来更像一个不可分割的整体，职责较为混乱。
- 系统扩展性低：增加一个新的对象需要在原有相关对象上增加引用，增加新的引用关系也需要调整原有对象，系统耦合度很高，对象操作很不灵活，扩展性差。
- 在面向对象的软件设计与开发过程中，根据“单一职责原则”，我们应该尽量将对象细化，使其只负责或呈现单一的职责。
- 对于一个模块，可能由很多对象构成，而且这些对象之间可能存在相互的引用，为了减少对象两两之间复杂的引用关系，使之成为一个松耦合的系统，我们需要使用中介者模式，这就是中介者模式的模式动机。

中介者模式***(Mediator Pattern)***定义：用一个中介对象来封装一系列的对象交互，中介者使各对象不需要显式地相互引用，从而使其耦合松散，而且可以独立地改变它们之间的交互。中介者模式又称为调停者模式，它是一种对象行为型模式。

案例

安理会做中介

```
//联合国机构
abstract class UnitedNations
{
    //声明
    public abstract void Declare(string message,Country colleagues)
}
```

国家类 相当于Colleague类

```
abstract class Country
{
    protected UnitedNations mediator;

    public Country(UnitedNations mediator)
    {
        this.mediator = mediator;
    }
}
```

美国类 相当于ConcreteColleague1类

```
class USA: Country
{
    public USA(UnitedNations mediator): base(mediator)
    {}
    //声明
    public void Declare(string message)
    {
        mediator.Declare(message, this);
    }

    //获得消息
    public void GetMessage(string message)
    {
        Console.WriteLine("美国获得对方消息" + message);
    }
}
```

伊拉克类 相当于ConcreteColleague2类

```
class Iraq: Country
{
    public Iraq(UnitedNations mediator): base(mediator)
    {}
    //声明
    public void Declare(string message)
    {
        mediator.Declare(message, this);
    }

    //获得消息
    public void GetMessage(string message)
    {
        Console.WriteLine("美国获得对方消息" + message);
    }
}
```

联合国安理会 相当于ConcreteMediator类

```
//联合国安全理事会
class UnitedNationsSecurityCouncil: UnitedNations
{
    private USA colleague1;
    private Iraq colleague2;

    //美国
    public USA Colleague1
    {
        set {colleague1 = value;}
    }

    //伊拉克
        public Iraq Colleague2
    {
        set {colleague2 = value;}
    }

    //声明
    public override void Declare(string message, Country colleague)
    {
        if(colleague == colleague1)
        {
            colleague2.GetMessage(message);
        }else
        {
            colleague1.GetMessage(message);
        }
    }
}
```

客户端调用

```
static void Main(string[] args)
{
    UnitedNationsSecurityCouncil UNSC = new UnitedNationsSecurityCo

    USA c1 = new USA(UNSC);
    Iraq c2 = new Iraq(UNSC);

    UNSC.Colleague1 = c1;
    UNSC.Colleague2 = c2;

    c1.Declare("不准研发核武器");
    c2.Declare("我们没有，但我们不怕侵略");

    Console.Read();
}
```

中介者模式的优点首先是Mediator的出现减少了各个Colleague的耦合,可以毒瘤地改变和复用各个Colleague类和Mediator。其次由于把对象如何协作进行类抽象，将中介作为一个独立的概念并且将其封装在一个对象中，这样关注的对象就从对象本身的行为转移到它们的交互上来，也就是站在一个更宏观的角度去看待系统。

但是，由于ConcreyeMediator控制了集中化，于是就把交互复杂性变为了中介者的复杂性，这就使得中介者会变得比任何一个ConcreteColleague都复杂。

享元模式(Flyweight Pattern)

简介

享元模式（英语：Flyweight Pattern）是一种软件设计模式。它使用共享物件，用来尽可能减少内存使用量以及分享资讯给尽可能多的相似物件；它适合用于当大量物件只是重复因而导致无法令人接受的使用大量内存。通常物件中的部分状态是可以分享。常见做法是把它们放在外部数据结构，当需要使用时再将它们传递给享元。

典型的享元模式的例子为文书处理器中以图形结构来表示字符。一个做法是，每个字形有其字型外观, 字模 metrics, 和其它格式资讯，但这会使每个字符就耗用上千字节。取而代之的是，每个字符参照到一个共享字形物件，此物件会被其它有共同特质的字符所分享；只有每个字符（文件中或页面中）的位置才需要另外储存。

示例

```
public enum FontEffect {
    BOLD, ITALIC, SUPERSCRIPT, SUBSCRIPT, STRIKETHROUGH
}

public final class FontData {
    /**
     * A weak hash map will drop unused references to FontData.
     * Values have to be wrapped in WeakReferences,
     * because value objects in weak hash map are held by strong re
     */
    private static final WeakHashMap<FontData, WeakReference<FontDa
        new WeakHashMap<FontData, WeakReference<FontData>>();
    private final int pointSize;
    private final String fontFace;
    private final Color color;
    private final Set<FontEffect> effects;

    private FontData(int pointSize, String fontFace, Color color, E
        this.pointSize = pointSize;
        this.fontFace = fontFace;
        this.color = color;
        this.effects = Collections.unmodifiableSet(effects);
    }

    public static FontData create(int pointSize, String fontFace, C
        FontEffect... effects) {
        EnumSet<FontEffect> effectsSet = EnumSet.noneOf(FontEffect.
        for (FontEffect fontEffect : effects) {
            effectsSet.add(fontEffect);
        }
        // We are unconcerned with object creation cost, we are rec
```

```
        FontData data = new FontData(pointSize, fontFace, color, ef

        // Retrieve previously created instance with the given valu
        WeakReference<FontData> ref = FLY_WEIGHT_DATA.get(data);
        FontData result = (ref != null) ? ref.get() : null;

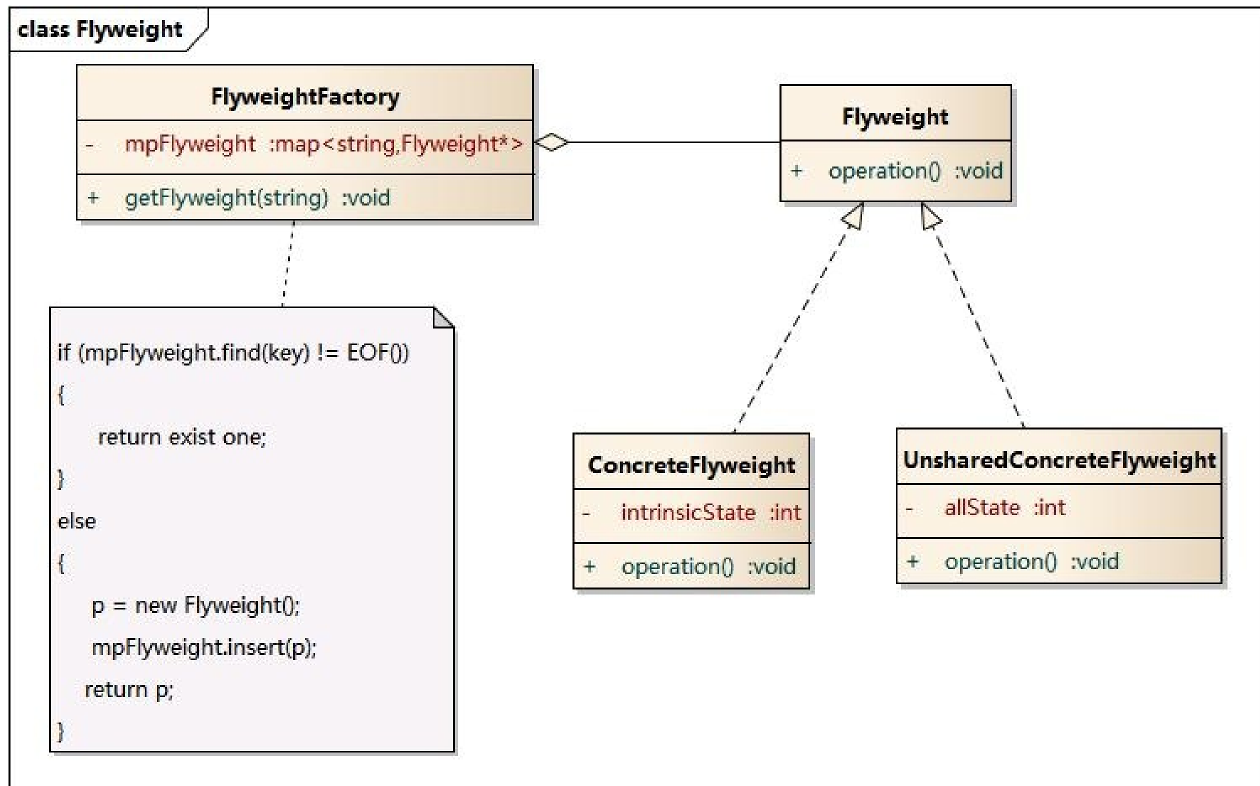
        // Store new font data instance if no matching instance exi
        if (result == null) {
            FLY_WEIGHT_DATA.put(data, new WeakReference<FontData> (
            result = data;
        }
        // return the single immutable copy with the given values
        return result;
    }

    @Override
    public boolean equals(Object obj) {
        if (obj instanceof FontData) {
            if (obj == this) {
                return true;
            }
            FontData other = (FontData) obj;
            return other.pointSize == pointSize && other.fontFace.e
                && other.color.equals(color) && other.effects.equal
        }
        return false;
    }

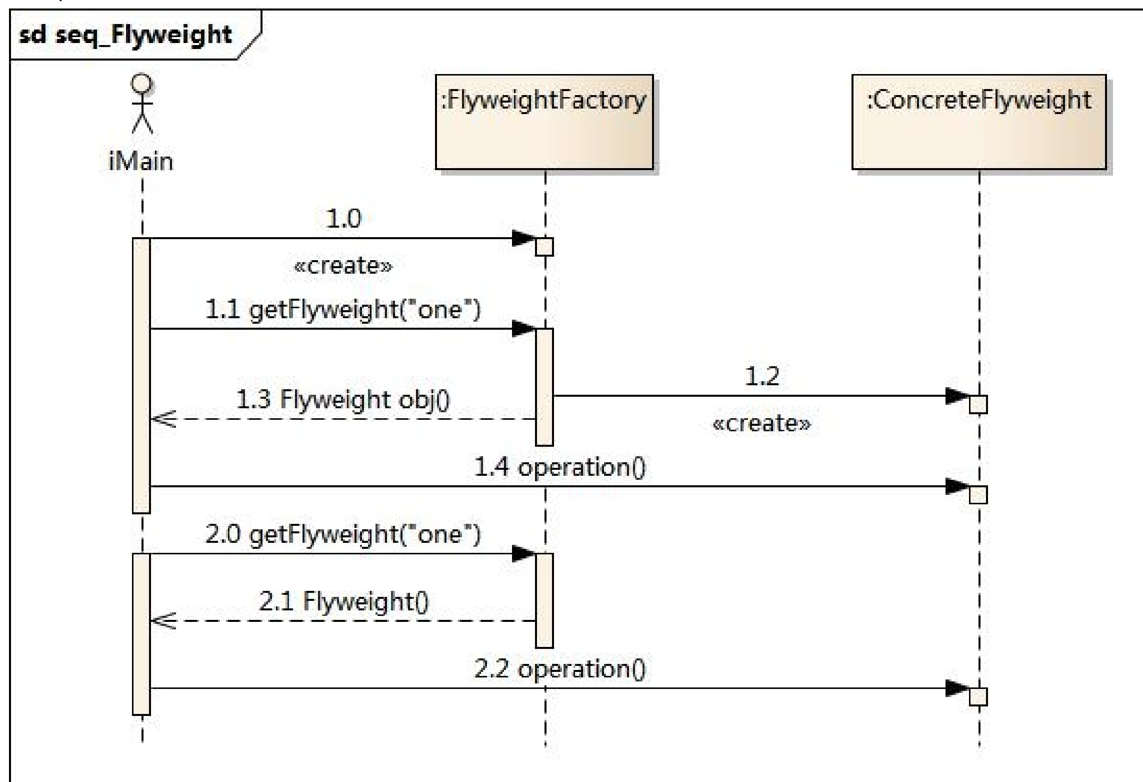
    @Override
    public int hashCode() {
        return (pointSize * 37 + effects.hashCode() * 13) * fontFac
    }

    // Getters for the font data, but no setters. FontData is immut
}
```

结构图



时序图



实例

第一版

```
//网站
class WebSite
{
    private string name = "";
    private WebSite(string name)
    {
        this.name = name;
    }
    public void Use()
    {
        Console.WriteLine("网站分类" + name);
    }
}
```

客户端代码

```
static void Main(string args)
{
    WebSite fx = new WebSite("产品展示");
    fx.Use();

    WebSite fy = new WebSite("产品展示");
    fy.Use();

    WebSite f1 = new WebSite("博客");
    f1.Use();

    WebSite f2 = new WebSite("博客");
    f2.Use();

    Console.Read();
}
```

第二版

网站抽象类

```
abstract class WebSite
{
    public abstract void Use();
}
```

具体网站类

```
class ConcreteWebSite :WebSite
{
    private string name = "";
    public ConcreteWebSite(string name)
    {
        this.name =name;
    }
    public override void Use()
    {
        Console.WriteLine("网站分类:" + name);
    }
}
```

网站工厂类

```
//网站工厂
class WebSiteFactory
{
    private HashTable flyweights = new Hashtable();

    //获得网站分类
    public WebSite GetWebSiteCategory(string key)
    {
        if (!flyweight.ContainKey(key))
        flyweights.Add(key, new ConcreteWebSite(key));
        return ((WebSite)flyweights[key]);
    }
    //获得网站分类总数
    public int GetWebSiteCount()
    {
        return flyweights.Count;
    }
}
```

客户端代码

```
static void Main(string[] args)
{
    WebSiteFactory f = new WebSiteFactory();

    WebSite fx = f.GetWebSiteCategory("产品展示");
    fx.Use();

    WebSite fy = f.GetWebSiteCategory("产品展示");
    fy.Use();

    WebSite f1 = f.GetWebSiteCategory("博客");
    f1.Use();

    WebSite f2 = f.GetWebSiteCategory("博客");
    f2.Use();

}
```

实际上，享元模式可以避免大量非常详细类的开销。在程序设计中，有时需要生成大量细力度的类实例来表示数据。如果能发现这些实例除了几个参数外基本上都是相同的，有时就能够受大幅度地减少需要实例化的类的数量。如果能把这些蚕食移到类实例的外面，在方法调用时将它们传递进来，就可以通过共享大幅度地减少单个实例的数目。

第三版

用户类，用于网站的客户端账号，是"网站"类的外部状态

```
//用户
public class User
{
    private string name;
    public User(string name)
    {
        this.name = name;
    }
    public string name
    {
        get {return name;}
    }
}
```

网站抽象类

```
abstract class WebSite
{
    public abstract void Use(User user);
}
```

具体网站类

```
class ConcreteWebSite: WebSite
{
    private string name = "";
    public ConcreteWebSite(string name)
    {
        this.name = name;
    }

    public override void Use(Use user)
    {
        Console.WriteLine("网站分类:" + name + "用户:" + user.name);
    }
}
```

网站工厂类

```
class WebSiteFactory
{
    private Hashtable flyweights = new HashTable();

    //获得网站分类
    public WebSite GetWebSiteCateegory(string key)
    {
        if (!flyweights.ContainsKey(key))
        flyweights.Add(key, new ConcreteWebSite(key));
        return ((webSite)flyweightskey[key]);
    }

    //获得网站分类总数
    public int GetWebSiteCount
    {
        return flyweights.Count;
    }
}
```

客户端代码

```
static void Main(string[] args)
{
    WebSiteFactory f = new WebSiteFactory();

    WebSite fx = f.GetWebSiteCategory("产品展示");
    fx.Use(new User("小菜"));

    WebSite fy = f.GetWebSiteCategory("产品展示");
    fy.Use(new User("小白"));

    WebSite f1 = f.GetWebSiteCategory("博客");
    f1.Use(new User("小黑"));

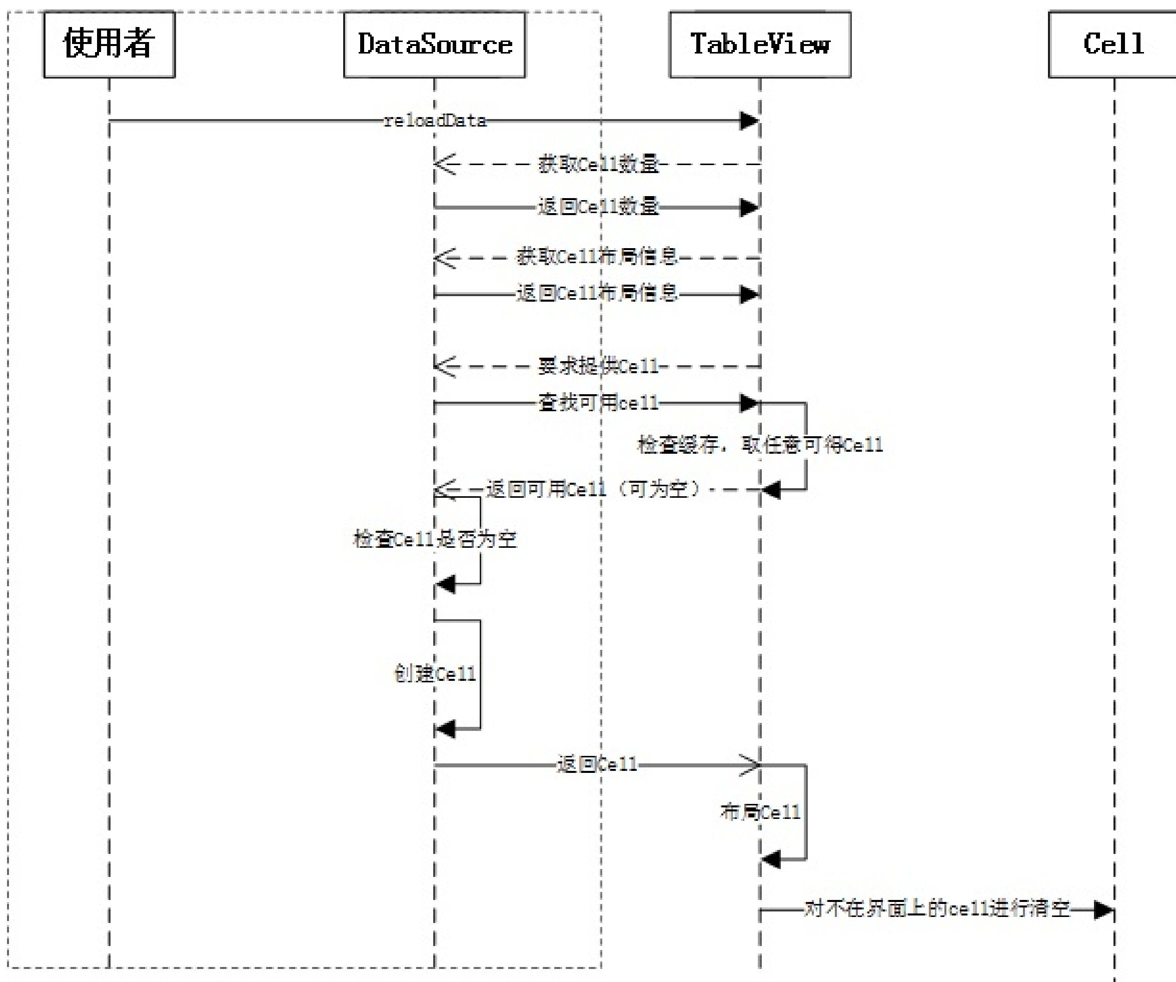
    WebSite f2 = f.GetWebSiteCategory("博客");
    f2.Use(new User("小绿"));

    Console.Read();
}
```

如果一个应用程序使用了大量的对象，而大量哒这些对象造成了很大的储存开销时就应该考虑使用；还有就是对象的大多数状态可以外部状态，如果删除对象的外部状态，那么可以用相对较少的共享对象取代很多组对象，此时可以考虑使用享元模式。

经典案例

Cell的重用



在使用UITableView的时候我们应该熟悉这样的接口：

```
- (id)dequeueReusableCellWithIdentifier:(NSString *)identifier

//ios6
- (id)dequeueReusableCellWithIdentifier:(NSString *)identifier forI
```

在要使用一个Cell的时候我们先去看看tableView中有没有可以重用的cell，如果有就用这个可以重用的cell，只有在没有的时候才去创建一个Cell。这就是享元模式。

享元模式可以理解成，当细粒度的对象数量特别多的时候运行的代价会相当大，此时运用共享的技术来大大降低运行成本。比较突出的表现就是内容有效的抑制内存抖动的情况发生，还有控制内存增长。它的英文名字是flyweight，让重量飞起来。哈哈。名副其实，在一个TableView中Cell是一个可重复使用的元素，而且往往需要布局的cell数量很大。如果每次使用都创建一个Cell对象，系统的内容抖动会非常明显，而且系统的内存消耗也是比较大的。突然一想，享元模式只是给对象实例共享提供了一个比较霸道的名字吧。

```
- (DZTableViewCell*) dzTableView:(DZTableView *)tableView cellAtRow
{
    static NSString* const cellIdentifiy = @"detifail";
    DZTypeCell* cell = (DZTypeCell*)[tableView dequeueDZTalbeViewCe
    if (!cell) {
        cell = [[DZTypeCell alloc] initWithIdentifiy:cellIdentifiy]
    }
    NSString* text = _timeTypes[row];
    return cell;
}
```

解释器模式(Interpreter Pattern)

简介

解释器模式:给定一个语言，定义它的语法的一种表示，并定义一个解释器，这个解释器使用该表示来解释语言中的句子。

解释器模式需要解决的是，如果一种特定类型的问题发生的频率足够高，那么可能就值得将该问题的各个实例表述为一个简单语言中的句子。这样就可以构建一个解释器，该解释器通过解释这些句子来解决该问题。

比如说，我们常常会在字符串中搜索匹配字符或判断一个字符串是否符合我们规定的格式，此时一般我们会用正则表达式匹配。解释器为正则表达式定义了一个文法，如何表示一个特定的正则表达式，以及如何解释这个正则表达式。

当有一个语言需要解释执行，并且你可以将该语言中的句子表示为一个抽象语法树时，可以使用解释器模式。

用了解释器模式，就意味着可以很容易地改变和扩展文法，因为该模式使用类来表示文法规则，你可以使用继承来改变或者扩展该文法。也比较容易实现文法，因为定义抽象语法树中各个节点的类的实现大体类似，这些类都易于直接编写。

解释器模式也有不足的，解释器模式为文法中的每一条规则至少定义了一个类，因此包含了许多规则的文法可能难以管理和维护。建议当文法非常复杂时，使用其它的技术如语法分析程序或编译器生成器来处理。

实例

音乐解释器

演奏内容类(context)

```
//演奏内容
class PlayContext
{
    //演奏文本
    private string text;
    public string PlayText
    {
        get {return text;}
        set {text = value;}
    }
}
```

表达式类(AbstractExpression)

```
abstract class Expression
{
    //解释器
    public void Interpret(PlayContext context)
    {
        if (context.PlayText.Length == 0)
        {
            return;
        }
        else
        {
            //此方法用于将当前演奏文本第一条命令获得命令字母和其参数值。例如"
            string playKey = context.Playtext.subText.subString(0,1
            context.PlayText = context. PlayText.subString(2);
            doublePlayValue = Convert.ToDouble(context.PlayText.Sub
            context.PlayText = context.PlayText.substring(context.P

            Excute(playKey, playValue);
        }
    }
    //执行
    public abstract void Excute(string key, doubel value);
}
```

音符类(TerminalExpression)

```
class Note: Expression
{
    public override void Excute(string key, double value)
    {
        string note ="";
        switch (key)
        {
            case: "C":
                note = "1";
                break;
            case: "D":
                note = "2";
                break;
            case: "E":
                note = "3";
                break;
            case: "F":
                note = "4";
                break;
            case: "G":
                note = "5";
                break;
            case: "A":
                note = "6";
                break;
            case: "B":
                note = "7";
                break;
        }
        Console.Write("{0}", note);
    }
}
```

音符类(TerminalExpression)

```
class Scale: Expression
{
    public override void Excute(string key, double value)
    {
        string scale = "";
        switch (Convert.ToInt32(value))
        {
            case 1:
                scale = "低音";
                break;
            case 2:
                scale = "中音";
                break;
            case 3:
                scale = "高音";
                break;
        }
        Console.Write("{0}",scale);
    }
}
```

客户端代码

```
static void Main(string[] args)
{
    PlayContext context = new PlayContext();
    //音乐-上海滩
    Console.WriteLine("上海滩:");
    context.PlayText = "O 2 E 0.5 G 0.5 A 3 E 0.5 G 0.5 D 3 E 0.5",
    Expression expression = null;
    try
    {
        while (context.PlayText.Length > 0)
        {
            string str = context.PlayText.Substring(0.1);
            switch (str)
            {
                case "O":
                    expression = new Scale();
                    break;
                case "C":
                case "D":
                case "E":
                case "F":
                case "G":
                case "A":
                case "A":
                case "B":
                case "P":
                    expression = new Note();
                    break;
            }
            expression.Interpret(context);

        }
    }
    catch (Exception ex)
    {
        Console.WriteLine(ex.Message);
    }
    Console.Read();
}
```

比如说 我现在要增加一个文法，就是演奏速度。

音速类

```
class Speed :Expression
{
    public override void Excute(string key, double value)
    {
        string speed;
        if (value < 500) {
            speed = "快速";
        } else if (value >= 1000)
        {
            speed = "慢速";
        } else {
            speed = "快速";
        }
        Console.Write("{0}",speed);
    }
}
```

客户端代码(局部)

```
    context.PlayText = "T 500 O 2 E 0.5 G 0.5 A 3 E 0.5 G 0.5 D 3 E
    Expression expression = null;
    try
    {
        while (context.PlayText.Length > 0)
        {
            string str = context.PlayText.Substring(0.1);
            switch (str)
            {
                case "O":
                    expression = new Scale();
                    break;
                    case "T":
                    expression = new Speed();
                    break;
                case "C":
                case "D":
                case "E":
                case "F":
                case "G":
                case "A":
                case "A":
                case "B":
                case "P":
                    expression = new Note();
                    break;
            }
            expression.Interpret(context);

        }
    }
    catch (Exception ex)
    {
        Console.WriteLine(ex.Message);
    }
    Console.Read();
}
```

访问者模式(Visitor Pattern)

简介

访问者模式是一种将算法与对象结构分离的软件设计模式。

这个模式的基本想法如下：首先我们拥有一个由许多对象构成的对象结构，这些对象的类都拥有一个accept方法用来接受访问者对象；访问者是一个接口，它拥有一个visit方法，这个方法对访问到的对象结构中不同类型的元素作出不同的反应；在对象结构的一次访问过程中，我们遍历整个对象结构，对每一个元素都实施accept方法，在每一个元素的accept方法中回调访问者的visit方法，从而使访问者得以处理对象结构的每一个元素。我们可以针对对象结构设计不同的实在的访问者类来完成不同的操作。

访问者模式使得我们可以在传统的单分派语言（如Smalltalk、Java和C++）中模拟双分派技术。对于支持多分派的语言（如CLOS），访问者模式已经内置于语言特性之中了，从而不再重要。

```
interface Visitor {
    void visit(Wheel wheel);
    void visit(Engine engine);
    void visit(Body body);
    void visit(Car car);
}

class Wheel {
    private String name;
    Wheel(String name) {
        this.name = name;
    }
    String getName() {
        return this.name;
    }
    void accept(Visitor visitor) {
        visitor.visit(this);
    }
}

class Engine {
    void accept(Visitor visitor) {
        visitor.visit(this);
    }
}

class Body {
    void accept(Visitor visitor) {
        visitor.visit(this);
    }
```

```
}

class Car {
    private Engine  engine = new Engine();
    private Body    body   = new Body();
    private Wheel[] wheels
        = { new Wheel("front left"), new Wheel("front right"),
            new Wheel("back left") , new Wheel("back right")  };
    void accept(Visitor visitor) {
        visitor.visit(this);
        engine.accept(visitor);
        body.accept(visitor);
        for (int i = 0; i < wheels.length; ++ i)
            wheels[i].accept(visitor);
    }
}

class PrintVisitor implements Visitor {
    public void visit(Wheel wheel) {
        System.out.println("Visiting " + wheel.getName()
                            + " wheel");
    }
    public void visit(Engine engine) {
        System.out.println("Visiting engine");
    }
    public void visit(Body body) {
        System.out.println("Visiting body");
    }
    public void visit(Car car) {
        System.out.println("Visiting car");
    }
}

public class VisitorDemo {
    static public void main(String[] args) {
        Car car = new Car();
        Visitor visitor = new PrintVisitor();
        car.accept(visitor);
    }
}
```

实例

男人和女人

'人'类，是'男人'和'女人'类的抽象类

```
abstract class person
{
    protected string action;
    public string Action
    {
        get {return action;}
        set {action = value};
    }
    //得到结论或者反应
    public abstract void GetConclusion();
}
```

"男人"类

```
class Man: Person
{
    //得到结论或反应
    public override void GetConclusion()
    {
        if (action == "成功")
        {
            Console.WriteLine("{0}{1}时，背后多半有一个伟大的女人",this
        }
        else if (action == "失败")
        {
            Console.WriteLine("{0}{1}时，"蒙头喝酒，谁也不用劝",this.Ge
        }
        else if (action == "恋爱")
        {
            Console.WriteLine("{0}{1}时，"凡事不懂也要装懂",this.GetTy
        }
    }

}
```

"女人"类

```
class Woman: Person
{
    //得到结论或反应
    public override void GetConclusion()
    {
        if (action == "成功")
        {
            Console.WriteLine("{0}{1}时，背后多半有一个不成功的男人",thi
        }
        else if (action == "失败")
        {
            Console.WriteLine("{0}{1}时，"泪眼汪汪，谁也劝不了",this.Ge
        }
        else if (action == "恋爱")
        {
            Console.WriteLine("{0}{1}时，"凡事懂也要装不懂",this.GetTy
        }
    }

}
```

客户端代码

```
static void Main(string[] args)
{
    IList<Person> persons = new List<Person>();

    Person man1 = new Man();
    man1.Action = "成功";
    persons.add(man1);
    Person woman1 = new Woman();
    woman1.Action = "成功";
    persons.add(woman1);

    Person man2 = new Man();
    man2.Action = "失败";
    persons.add(man2);
    Person woman2 = new Woman();
    woman2.Action = "失败";
    persons.add(woman2);

    Person man3 = new Man();
    man3.Action = "恋爱";
    persons.add(man3);
    Person woman3 = new Woman();
    woman3.Action = "恋爱";
    persons.add(woman3);

    foreach (Person item in persons)
    {
        item.GetConclusion();
    }

    Console.Read();

}
```

这个算是面向对象编程，但是'男人'和'女人'类当中的那些if....else....很是碍眼。而且如果我们增加一个'结婚状态',怎么改？

用模式实现

'状态'的抽象和'人'的抽象

```
abstract class Action
{
    //得到男人结论或反应
    public abstract void GetManConclusion(Man concreteElementA);
    public abstract void GetWomanConclusion(Woman concreteElementB
}

abstract class Person
{
    //接受
    public abstract void Accept(Action visitor);
}
```

具体"状态"类

```
//成功
class Success: Action
{
    public override void GetManConclusion(Man concreteElementA)
    {
        Console.WriteLine("{0}{1}时，背后多半有一个伟大的女人",this.Get
    }

    public abstract void GetWomanConclusion(Woman concreteElementB)
    {
        Console.WriteLine("{0}{1}时，背后多半有一个不成功的男人",this.Ge
    }
}

//失败
class Falling: Action
{
    public override void GetManConclusion(Man concreteElementA)
    {
        Console.WriteLine("{0}{1}时，"蒙头喝酒，谁也不用劝",this.GetTyp
    }

    public abstract void GetWomanConclusion(Woman concreteElementB)
    {
        Console.WriteLine("{0}{1}时，"泪眼汪汪，谁也劝不了",this.GetTyp
    }
}

//恋爱
class Amativeness: Action
{
    public override void GetManConclusion(Man concreteElementA)
    {
        Console.WriteLine("{0}{1}时，"凡事不懂也要装懂",this.GetType()
    }

    public abstract void GetWomanConclusion(Woman concreteElementB)
    {
        Console.WriteLine("{0}{1}时，"凡事懂也要装不懂",this.GetType()
    }
}
```

"男人"类和"女人"类

```
//男人
class Man : Person
{
    public override void Accept(Action visitor)
    {
        //***
        visitor.GetManConclusion(this);
    }
}


//女人
class Woman : Person
{
    public override void Accept(Action visitor)
    {
        //***
        visitor.GetWomanConclusion(this);
    }
}
```

这里需要讲下双派技术，首先客户端将具体状态作为参数传递给"男人"完成了一次分派，然后男人类调用方法"男人反应",同时将自己作为参数传递进去。这便完成了第二次分派。双分派意味着得到执行的操作决定于请求的种类和两种接受者的类型。"接受"方法就是一个双分派的操作，它得到执行的操作不仅决定于'状态'类的具体状态，还决定于它访问的'人'的类别。

```
//对象结构
class ObjectStructure
{
    private IList<Person>elements = new List<Person>();

    //增加
    public void Attach(Person element)
    {
        elements.Add(element);
    }

    //移除
    public void Detach(Person element)
    {
        elements.Remove(element);
    }

    //查看显示
    public void Display(Action visitor)
    {
        foreach(Person e in elements)
        {
            e.Accept(visitor);
        }
    }
}
```

客户端代码

```
static void Main(string[] args)
{
    ObjectStructure o = new ObjectStructure();
    o.Attach(new Man());
    o.Attach(new Man());

    //成功时反应
    Success v1 = new Success();
    o.Display(v1);

    //失败时反应
    Success v1 = new Success();
    o.Display(v1);

    //恋爱时反应
    Success v1 = new Success();
    o.Display(v1);

    Console.Read();

}
```

访问者适合于数据结构相对稳定的系统。它把数据结构和作用于结构之上的操作之间的耦合解脱开来，使得操作集合可以相对自由地*y*演化。

访问者模式的目的是要把处理从数据结构分离出来。如果这样的系统有比较稳定的数据结构，又有易于变化的算法的话，使用访问者模式就是比较适合的，因为访问者模式使得算法操作的增加变得容易。

那其实访问者模式的优点就是增加新的操作很容易，因为增加新的操作就意味着增加一个新的访问者,访问者模式将有关的行为集中到一个访问者对象中。

那访问者的缺点其实也就是使得增加新的数据结构变的困难了。

正如《设计模式》的作者GoF对访问者模式的描述：大多数情况下，你并需要使用访问者模式，但是当你一旦需要使用它时，那你就是眞的需要它了。当然这只是针对眞正的大牛而言。在现实情况下（至少是我所处的环境当中），很多人往往沉迷于设计模式，他们使用一种设计模式时，从来不去认眞考虑所使用的模式是否适合这种场景，而往往只是想展示一下自己对面向对象设计的驾驭能力。编程时有这种心理，往往会发生滥用设计模式的情况。所以，在学习设计模式时，一定要理解模式的适用性。必须做到使用一种模式是因为了解它的优点，不使用一种模式是因为了解它的弊端；而不是使用一种模式是因为不了解它的弊端，不使用一种模式是因为不了解它的优点。